OPTIMAL PURCHASING STRATEGY FOR BANDWIDTH: A BUYER'S PERSPECTIVE

by

Frances Sharer Vergara

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Information Technology

Charlotte

2008

Approved by:

_____
Dr. Moutaz Khouja

_____
Dr. Ram Kumar

_____
Dr. Cem Saydam

_____
Dr. Sungjune Park

_____
Dr. Farid Tranjan

ABSTRACT

FRANCES SHARER VERGARA. Optimal purchasing strategy for bandwidth: A buyer's perspective. (Under direction of DR. MOUTAZ KHOUJA)

In today's burgeoning business environments, data frenetically pulses through massive interconnected networks as firms create, compile, and disseminate vast amounts of information. Organizations need to acquire the bandwidth that allows them to effectively share information. To acquire bandwidth, a firm agrees to pay a provider a fee, and the provider agrees to provide an acceptable quality bandwidth.

Price of bandwidth depends on two parameters: size and duration of contracts. Size is the amount of bandwidth purchased, and duration is the length of the contract. Bandwidth prices have been declining over time. Therefore, managers deciding on contracts to cover a given planning horizon have to decide on the number of contracts to purchase, their bandwidth (size), and their lengths (duration) to minimize costs. This research analyzes bandwidth contracting decisions from a buyer's perspective. Historical data will be used to estimate bandwidth cost as a function of contract size and duration at a point in time, and to estimate bandwidth cost overtime. Both mathematical programming and evolutionary algorithms will be used to solve the problem under deterministic increasing bandwidth demand. In addition, simulation will be used with evolutionary algorithms to solve the problem under stochastic demand and prices.

ACKNOWLEDGEMENTS

In completing this study, there were many people involved in the process. I would like to acknowledge and extend my deep appreciation to them.

First, I would like to thank the members of my dissertation committee: Dr. Moutaz Khouja for being a mentor, advocate and a guide throughout the dissertation process, Dr. Cem Saydam for his critical inputs and feedback on this study, Dr. Ram Kumar and Dr. Sungjune Park for their advice and support, and Dr. Farid Tranjan for his assistance with the dissertation process.

I would like to thank my husband Hari K Rajagopalan, for his love, patience and support during the last few years. It was always nice to have someone who could and would talk to you about your dissertation late at night and show you a different way of looking at the same problem.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

Bandwidth Contract Problem (BCP)

Deterministic Bandwidth Contract Problem (DBCP)

Stochastic Bandwidth Contract Problem (SBCP)

Evolutionary Algorithm (EA)

Deterministic Bandwidth Contract Problem-Evolutionary Algorithm (DBCP-EA)

Stochastic Bandwidth Contract Problem-Evolutionary Algorithm (SBCP-EA)

Model Notation:

T = Length of planning horizon,

τ = Minimum duration of a contract, e.g., 1 month,

N = Number of contract periods in the planning horizon, $N = T/\tau$,

i = Beginning period index for a contract, $i = 1,2 \ldots N$,

j = Ending period index for a contract, $j = 2 \ldots N+1$,

D_i = Demand for bandwidth in period i,

$x_{i,j} = \begin{cases} 1 \text{ if a contract beginning at period } i \text{ and ending at the beginning of period } j \text{ is purchased,} \\ 0 \text{ otherwise,} \end{cases}$

$P_{i,j}$ = Price per Mbps per time unit for contract beginning in period i and ending at the beginning of period j; a function of size (Q) and duration (L),

$Q_{i,j}$ = Size of a contract that begins in period i and ends at the beginning of period j,

$L_{i,j} = j - i$, the duration of a contract that begins in period i and ends at the beginning of period j,

g_D = Demand growth rate,

$r_D = 1 + g_D$,

a_D = Initial demand at the beginning of the planning horizon,

g_P = Price decrease rate,

$r_P = 1 - g_P$, and

a_P = Initial bandwidth price at the beginning of the planning horizon.

CHAPTER 1 INTRODUCTION

In today's burgeoning business environments, data frenetically pulses through massive interconnected networks as firms create, compile, and disseminate vast amounts of information. Organizations need to acquire the resources that allow them to effectively share information. Just as networks have developed to enable the distribution of electricity, or facilitate telephone conversations, networks have also developed to enable the transmission of digital data from one location to another. These networks can be dedicated channels or switched channels. A dedicated channel is a link that is permanently established. In general, a dedicated channel is more secure, but also more expensive. These networks can be privately owned, or leased from a third party. A switched channel dynamically establishes a path between two points based on present network conditions. Links to these types of networks are often the most cost effective alternative. In a switched network, data, containing the address of the desired destination, is placed on a network. The destination address is used to guide the data through the network. As the data traverses the network, a communications path from source to destination could span a variety of links, and each link could have a different bandwidth. The bandwidth for the entire path is limited by the smallest bandwidth within the path. The link in the path with the smallest bandwidth is referred to as a bottleneck (http://compnetworking.about.com/library/glossary/bldef-bandwidth.htm?terms =bandwidth).

Typically, a firm seeking to purchase bandwidth from a vendor engages in a contractual agreement in which the firm agrees to pay a fee, and the provider agrees to provide a set bandwidth for a given period of time at a specified level of quality.

1.1 What is Bandwidth?

Bandwidth is a measure of how much data can be transmitted through a network, either through a guided media, such as a wire or cable, or an unguided media, which uses the atmosphere to propagate a transmission. In other words, bandwidth is a measure of the capacity, or data transfer rate, of a link. It is sometime helpful to think of bandwidth as a pipe. The larger the pipe, the more data can be sent. Bandwidth is also known as throughput (http://compnetworking.about.com/library/glossary/bldef-andwidth.htm?terms =bandwidth).

In digital transmission, bandwidth is measured as the number of bits, or binary digits that can be transmitted through a circuit or channel within a given period of time. The measure of bandwidth is recorded in bits per second (bps) (http://compnetworking.about.com/library/glossary/bldef-bandwidth.htm?terms =bandwidth). One thousand bps is equivalent to 1 Kbps (kilobit per second), one thousand Kbps (or one million bps) is equivalent to 1 Mbps (Megabit per second), one thousand Mbps (or one billion bps) is equivalent to 1 Gbps (Gigabit per second), and one thousand Gbps (or one trillion bps) is equivalent to 1 Tbps (Terabit per second) (Horak, 2000).

1.1.1 Common Digital Communication Media

There are numerous transmission media and technologies available. The transmission mediums include guided media, such as Twisted Pair, Coaxial Cable, Fiber Optic, and unguided, or wireless, media, such as Microwave Radio and Satellite.

Twisted pair, as its name suggests, consist of a pair of twisted copper wire that is used to propagate a carrier wave signal from one destination to another. Twisted pair is the oldest transmission medium (Laudon and Laudon, 2003), and has historically been used by telephone companies for the transmission of analog voice signals; however digital data can also be transmitted over twisted pair. The advantage of twisted pair is that it is ubiquitous and inexpensive. The disadvantage is that the transmission speed is somewhat limited and is significantly less than some of the other media. Some of the technologies employed for the transmission of digital data over twisted pairs include T-1 connections, Category 5 (Cat 5) copper, and Asymmetric Digital Subscriber Loop (ADSL) (Horak, 2000).

A T-1 connection can be used to transmit data and voice and has a capacity of 1.544 Mbps (1,544,000 bits per second) (FitzGerald, and Dennis, 1999). T-1 circuits are often used for inverse multiplexing. Multiplexing is a process where multiple signals are merged onto one channel for transmission. Inverse multiplexing is the reverse of multiplexing, for inverse multiplexing a single signal is split up and placed onto two or more channels for transmission. The end result, a single signal is able to traverse the network in a fraction of the time it would take if the signal had traveled over a single channel (FitzGerald et al., 1999). T-1 circuits are the most frequently used digital line in the United States, Canada, and Japan (http://searchnetworking.techtarget.com/sDefinition

/0,,sid7_gci213084,00.html). Category 5 is a network cable standard for twisted pair. These cables are often used in 100Base-T Ethernet networks and provide a maximum data rate, or bandwidth, of 100 Mbps at a maximum distance of 200 meters (FitzGerald et al., 1999). An Asymmetric Digital Subscriber Line (ADSL) is a technology that uses twisted pair media to transmit digital data at high bandwidths. The ADSL provides a continuous circuit and is comprised of three separate channels (http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci213764,00.html). One channel is a dedicated analog circuit for telephone voice transmission. The remaining two channels are used for incoming (downstream) and outgoing (upstream) digital data transmissions. In ADSL, these two channels are asymmetric, meaning that one channel has a much higher bandwidth than the other channel. The rational for this configuration is that often incoming data bandwidth requirements are greater than outgoing data bandwidth requirements. A limitation to ADSL is distance. For ADSL to function, the user needs to be located no more than three miles from a telephone companies end office; with better transfer rates as the user moves closer (FitzGerald et al., 1999).

The second media for digital data transmission is coaxial cable which is a guided media like twisted pair; however the construction of coaxial cable is quite different than that of twisted pair. A coaxial cable is comprised of two conductors. The first conductor is a cylindrical copper core which is located at the center of the coaxial cable. This core conductor is surrounded by a thick cylinder of insulating material. Surrounding this insulating material is a second conductor which resembles a wire mesh. Finally, surrounding the wire mesh is a secondary insulating material (FitzGerald et al., 1999). Two common network cable standards related to coaxial cables include RG-58 (10Base-2

Ethernet) and RG-8 (10Base-5 Ethernet). Both standards have a maximum data transmission rate of 10 Mbps, and a maximum distance of 185 meters and 500 meters, respectively (FitzGerald et al., 1999).

Fiber optics, another guided medium, is a relatively new media. A fiber optic cable transmits data in the form of pulses of light within extremely thin glass or plastic filaments, or fibers (FitzGerald et al., 1999). The original signal is an electrical signal which is converted into an optical (light) signal. Once the signal reaches its destination, it is converted back to an electrical signal.

Since the cost of laying fiber is high and demand is uncertain, new technologies for the effective use of bandwidth such as dense wavelength division multiplexing (DWDM) are emerging (http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci 213892,00.html). With the use of DWDM, multiple signals can be sent over one channel at the same time, in this case a single fiber optic filament, using a different color of light for each individual signal. Each color of light is a separate frequency wavelength, and for this reason each signal remains separate and intact when it reaches its destination. The multiple light signals, upon reaching the destination, are then demultiplexed back into individual signals (Laudon, et al., 2003). Prior to the development of DWDM only one signal of light (i.e., a single wavelength) could be transmitted over a single fiber (Laudon et al., 2003). With the advent of DWDM the capacity of each fiber optic filament is significantly increased. DWDM began by transmitting four separate light wave frequencies, each frequency comprising a separate stream of data, and each propagating through the fiber at approximately 10 Gbps, for a total bandwidth of 40 Gbps over a single fiber. Thirty-two light wave frequencies have been transmitted over a

single fiber at a propagation rate of approximately 10 Gbps resulting in a total bandwidth of approximately 320 Gbps per fiber (Horak, 2000). It is speculated that DWDM has the capability to transmit up to 160 separate signals, i.e., separate wavelengths, over a single fiber for a total transmission speed of up to 6.4 Tbps, or 6.4 trillion bits per second (Laudon et al., 2003). Since optical fibers are often bundled into cables that contain thousands of single fibers, the potential bandwidth of fiber optical networks is astronomical (Horak, 2000).

The advantage of fiber optics over traditional guided media is that fiber optic cables are able to transmit data at a much higher rate. Fiber optic cables can also carry data, voice, graphical, and video transmission at the same time (http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci212113,00.html). This means that a variety of data formats can be accommodated simultaneously (http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci213892,00.html). In addition, fiber optical cables, since they do not transmit electrical signals, are not affected by electromagnetic disturbances that can degrade the quality of the data being transmitted, as can happen with some of the other guided media. Fiber optic cables are also much less susceptible to the negative effects of environmental conditions (FitzGerald et al., 1999).

Microwave radio is a type of unguided transmission media that employs terrestrial transmission stations to propagate a signal over some distance. It supports high-volume, long-distance, point-to-point data transmission (Horak, 2000; Laudon et al., 2003). Bandwidth provided by microwave transmission can be greater than 6 Gbps. The disadvantage of microwave is that the frequencies within the microwave radio spectrum

are limited, and these frequencies are licensed to users based on geographical regions. Once a frequency is licensed, it becomes unavailable for other uses within that regional area (Horak, 2000).

Satellites also use microwave signals to transmit data, however, for satellite transmissions the satellite(s) are orbiting transmission relay stations, which perform the same function as the terrestrial transmission stations do for microwave transmissions. Signals are transmitted from terrestrial stations to the satellite and then sent back to Earth to another terrestrial station, or stations. Satellites can broadcast transmissions over a large area so that many terrestrial stations have the ability to receive a broadcast signal. Satellites employ a single uplink station and one or more downlink stations. Therefore, because of the large footprint (the area on the ground where the signal is accessible), satellites are ideal for point-to-multipoint network, such as television signal transmission (Horak, 2000). Today's satellites handle thousands of signals. In addition, satellites can be employed to transmit a variety of signals, from simple data to broadcast television signals (http://searchmobilecomputing.techtarget.com/sDefinition/0,,sid40_gci212939,00.html).

One disadvantage of satellites is propagation delay, which is a time lag experienced due to the time required for a signal to travel from Earth to a satellite several thousand miles away. Satellites, while slower than microwave radio, are able to transmit data efficiently over immense distances, and into areas where it would be difficult to place physical wires and cables, or terrestrial transmission stations.

1.1.2 Media and Technology Selection Criteria

There are several factors to consider when selecting a transmission media. One factor is the type, or geographic scope, of the network to be used, since some media are predominantly used in certain types of networks. For example, microwave radio and satellites are usually used for WAN (wide area networks), while twisted pair, and coaxial cable are not. Fiber optic cables, on the other hand, are very versatile and can be used for various types of networks, either large or small. Transmission distance is another important factor to consider. Guided media, with the exception of fiber optic cables, must incorporate repeaters to boost the signal if it needs to travel more than a small distance. Twisted pair and coaxial cable can transmit data approximately 100 meters, and 500 meters, respectively, before a repeater is required. Optical fibers can transmit data as much as 75 miles before the signal degrades enough that it requires regeneration. It is expected that in the future, distances could reach 600 miles or more before a fiber optic signal will need to be regenerated. Transmission speeds, or bandwidth, inherent to each media is also important. Typically, twisted pair and coaxial cable have bandwidths ranging from 1 to 100 Mbps, while the wireless media, microwave radio and satellite, have bandwidths ranging from 20 to 50 Mbps, and fiber optic cable has bandwidths ranging from 100 Mbps to 10 Gbps. Yet another factor to consider is the level of error rate inherent to each media. Unguided media, or wireless media, are media most susceptible to transmission errors. For guided media, twisted pair has the highest error rate; and yet still has a lower error rate than unguided media. Fiber optic cable has the lowest error rate across all media, and coaxial cable falls somewhere between twisted pair

and fiber optic cable (FitzGerald et al., 1999). Data throughput is affected by both transmission speed and error rate (Horak, 2000).

Security is another important factor to consider. Since signals are propagated through the air for unguided or wireless media, transmissions are much less secure than over guided media. Fiber optic cables are the most secure medium. Finally, cost is a very important factor. Cost refers to the cost of the media itself, which is an important consideration for an entity building a network. Therefore, cost in this instance is from a provider/seller's point of view. Later, we will focus on the cost of purchasing bandwidth services, which is cost from the purchaser/buyer's perspective. For guided media, twisted pair, being the oldest and least sophisticated, is typically the cheapest transmission media, followed by coaxial cable, then fiber optic cable. Unguided media's cost is related to distance. Therefore, microwave radio is cheaper than satellite (FitzGerald et al., 1999). Table 1-1 reports relative information about each of the media discussed with respect to the various selection factors identified.

Table 1-1: Media Summary

Guided Media						
Media	**Network Type**	**Transmission Distance**	**Speed**	**Error Rates**	**Security**	**Cost**
Twisted Pair	LAN	Short	Low-High	Low	Good	Low
Coaxial Cable	LAN	Short	Low-High	Low	Good	Moderate
Fiber Optics	Any	Moderate-Long	High-Very High	Very Low	Very Good	High
Unguided Media						
Media	**Network Type**	**Transmission Distance**	**Speed**	**Error Rates**	**Security**	**Cost**
Microwave	WAN	Long	Moderate	Low-Moderate	Poor	Moderate
Satellite	WAN	Long	Moderate	Low-Moderate	Poor	Moderate

Source: (FitzGerald et al., 1999)

Finally, networks can also be classified as either analog or digital. There are numerous advantages to transmitting data digitally. One advantage is that it is very easy to compress digital data, and security, and error detection and retrieval are much better for digital networks. In addition, digital networks are easier to manage and upgrade (Horak, 2000).

1.2 The Demand for Bandwidth

Significant advances in information technology have brought about increased use of multimedia and Internet applications. As a result, many firms, including Internet service providers (ISPs), application service providers (ASPs), and companies with operations in multiple locations spread across one or more continents have seen their demand for telecommunications capacity increase significantly over the last decade.

It is estimated that more than half a trillion dollars of US-based company revenues in 1999 were generated through Internet activities (Galbi, 2001). It is not surprising that the demand for bandwidth grew throughout the 1990s (Galbi, 2001).

Galbi (2001) reports that Internet backbone traffic grew 1,000% per year in 1995 and 1996, but the growth rate fell to 100% per year in 1997 and 1998. The total Internet backbone bandwidth in the mid-1998 is estimated to be 110 Gbps (Galbi, 2001).

The deployment of fiber optic cables peaked around 1990 with a growth rate of 28%, but has continued to grow at a considerable rate (Galbi, 2001). The inter-office non-switched bandwidth sold by Regional Bell Operation Company (RBOC) grew on average 37.4% per year between 1989 and 1999 (Galbi, 2001). Furthermore, bandwidth data for combined US trans-Atlantic and trans-Pacific companies show that the growth rate from 1989 to 1995 was 981%, and from 1995 to 2000 was 4,215% (Galbi, 2001).

1.2.1 Bandwidth as a Commodity

Bandwidth is typically traded in discrete capacities, and international standards (http://www.iec.org/online/tutorials/sonet_trans/topic05.html) are often used, e.g., SONET. SONET (Synchronous Optical NETwork) is a fiber technology standard (Horak, 2000). Line transmission rates in SONET include 51 Mbps, 155 Mbps, and 622 Mbps, 2.5 Gbps, and 10 Gbps (Laudon et al., 2003). With the advances that have developed in optical technology and increasing demands for fast, secure, and accurate bandwidth services, fiber optics will undoubtedly be a major medium well into the future.

Of the countless networks that merge together to form a conglomerate network, e.g., the Internet and other communication networks, each individual network remains a separate entity made up of its own media and protocols. This means that a signal traveling through a series of node-to-node connections within a constellation of networks could travel over a variety of media and network types. Due to the enormous number of possible paths within the network, price competition among links is likely to be very low (Galbi, 2001). Consequently, there has been increasing effort by third-party vendors to create bandwidth exchanges so that bandwidth can be easily and effectively traded just as other commodities are traded, e.g., electricity. Currently, the transaction cost of bandwidth is high due to the fragmented nature of the individual networks (Galbi, 2001). While there is a push towards bandwidth exchanges, presently exchanges capture only a small sector of bandwidth sales (Galbi, 2001). "One knowledgeable industry observer, while calling past price trends a 'great anomaly', predicts that data transmission prices 'are likely to start a rapid decline soon.' The basis for this optimism seems to be that commodity markets for bandwidth will develop rapidly" (Galbi, 2001).

In fact, upon examination of prices for exchange-traded bandwidth it is found that these prices are significantly less than bandwidth prices offered outside exchanges (Galbi, 2001). Therefore, as bandwidth exchanges become more and more prevalent, bandwidth will become more standardized; as is the case with any commodity. In addition, as bandwidth technology expands and evolves it is reasonable to assume that prices will decline. According to Cheliotis (2000), in the late nineties a huge supply of bandwidth became available. "This trend will urge suppliers and consumers to engage more actively in trading excess capacity**,** leading to the formation of bandwidth commodity markets" (Cheliotis, 2000).

Demand uncertainty, technology uncertainty, and cost pressures on prices of bandwidth due to new entrants (Borthick, 2001) significantly increase the risk associated with telecommunications capacity planning for telecommunications companies. Demand uncertainty causes organizations to lay telecommunications capacity in two steps. In the first step, organizations lay excess capacity in the form of unlit optic fiber, or dark fiber. Later, when demand increases, an additional cost is incurred to "light" these dark fibers, thereby increasing bandwidth capacity.

The high degree of risk associated with telecommunications capacity planning decisions has led to the emergence of bandwidth intermediaries and marketplaces where bandwidth can be traded. One type of intermediary is wholesalers who typically enter into contracts with large telecommunications companies and manage the economic and technology risks associated with providing access to retail customers. Cheliotis (2000) relates that a bandwidth broker performs typical brokerage functions (i.e., providing trust, aggregation, and negotiation), in addition to functions that are specific to the procurement

of bandwidth. These functions include "trading multiple contracts in a spot market and even offering new composite contracts to customers, thus creating a secondary market for bandwidth" (Cheliotis, 2000). Other types of intermediaries include market makers who match bandwidth sellers with bandwidth consumers (http://www.bandwidthfinders.com/,; http://www.band-x.com/en/). These intermediaries cater to the needs of bandwidth providers seeking to sell excess bandwidth as well as consumers requiring bandwidth.

As stated above, it is thought that bandwidth will eventually evolve into a commodity, which will be traded through exchanges. However, two important developments are required for bandwidth to become a true tradable commodity. First, it is necessary to develop standardized contracts (with strict penalties for non-performance) that ensure that a communications path with a particular bandwidth between a source and destination provided by supplier A is interchangeable in terms of performance and quality with a similar offering from supplier B. Second, the infrastructure should provide inter-organizational connectivity between carriers, referred to as pooling points. The most cost effective route between two locations (say New York and LA) might involve multiple bandwidth suppliers, for example, New York to Chicago using supplier A, and Chicago to LA using supplier B. However, this combination of links from multiple carriers would work only if A and B can connect to each other's networks. Some organizations have invested in pooling points, and additional developments are underway. Current bandwidth markets, however, are only partially commoditized (Borthick, 2001; Cheliotis, 2000).

1.2.2 Pricing of Bandwidth

The cost of telecommunication capacity depends on the total bandwidth purchased, its geographical location, and the duration of the contract. Bandwidth prices must be viewed at both a single point in time, and over time. At any given point in time, contract prices per unit time decrease as size and duration of the contract increase. Over time, prices are also decreasing. Therefore, purchase decisions must consider the cost of being locked into an extended contract at a fixed price, when prices are likely to decrease over time.

Table 1-2 illustrates the annual prices for bandwidth at different capacities (sizes) and for different contract lengths provided by a bandwidth supplier (www.thecomputerking.com) at a single point in time. These prices are a snapshot of bandwidth prices from one supplier on December 3, 2001. Other web sites containing information regarding telecommunications pricing include band-x.com (http://www.band-x.com/en/) and telegeography.com (http://www.telegeography.com/). Under current market conditions characterized by excess capacity and increased competition, prices may be lower, but discounts for larger transmission rates and longer contracts are still available. Hence, the data in Table 1-2 is useful for research purposes.

Table 1-2: Illustrative Annual Prices for Bandwidth

		Contract Duration		
		Column (a)	Column (b)	Column (c)
		1 year*	2 years‡	3 years†
Bandwidth Size	6 Mbps	$14,310	$13,595	$12,164
	9 Mbps	$15,429	$14,658	$13,115
	12 Mbps	$16,548	$15,721	$14,066
	15 Mbps	$17,667	$16,784	$15,017
	18 Mbps	$18,786	$17,847	$15,968
	21 Mbps	$19,905	$18,910	$16,919
	45 Mbps	$25,500	$24,225	$21,675

Source: (www.thecomputerking.com)

*Column (a) — price per bandwidth size for a 1 year contract
‡Column (b) — price per bandwidth size for a 2 year contract, calculated as 95% of column (a)
†Column (c) — price per bandwidth size for a 3 year contract, calculated as 85% of column (a)

Table 1-3 provides a view of bandwidth pricing over time. Table 1-3 shows the U.S. local inter-office circuit prices in dollars per month per Mbps for four different circuits (VG, DDS, DS1, and DS3) from 1990 to 2000. These four types of circuits represented more than 80% of RBOC leased-line revenue in the 1990s (Galbi, 2001).

Table 1-3: US Local Inter-Office Circuit Prices (Dollars per Month per Mbps)

Year	VG (64,128 Kbps)	DDS (56 Kbps)	DS1 (1.5 Mbps)	DS3 (44.7 Mbps)
1990	159	2,514	191	16.17
1991	156	2,421	174	19.03
1992	140	2,092	177	18.07
1993	121	1,399	136	16.21
1994	134	971	123	17.95
1995	140	1,171	114	17.15
1996	144	900	116	16.53
1997	138	846	118	16.46
1998	149	925	114	16.73
1999	147	942	113	17.49
2000	143	878	112	17.17

Source: (Galbi, 2001)

Table 1-4 reports U.S. long-distance leased line prices for T-1 and T-3 circuits from 1994 through 1999. The data in Table 1-4 shows that prices declined by 20% between 1994 and 1999, for both types of circuits (Galbi, 2001).

Table 1-4: US Long-Distance Leased Line Prices

Year	T-1 (1.54 Mbps)	T-3 (44.74 Mbps)
1994	760	5,830
1995	760	5,830
1996	730	5,554
1997	680	5,260
1998	650	5,000
1999	620	4,750

Source: (Galbi, 2001)

This long-term decline in bandwidth prices is due to technological advances and excess capacity buildup by existing telecommunications companies in anticipation of increased demand, as well as new entrants. Some researchers expect this trend to continue as carriers with excess capacity are under pressure to sell this capacity through brokers and exchanges (Borthick, 2001; Cheliotis, 2000).

Figure 1-1 shows the price indices for Band-x starting from October 1998. As can be seen from Figure 1, from October 1998 to July 2000, Band-X's price index fell approximately 63% (Galbi, 2001).

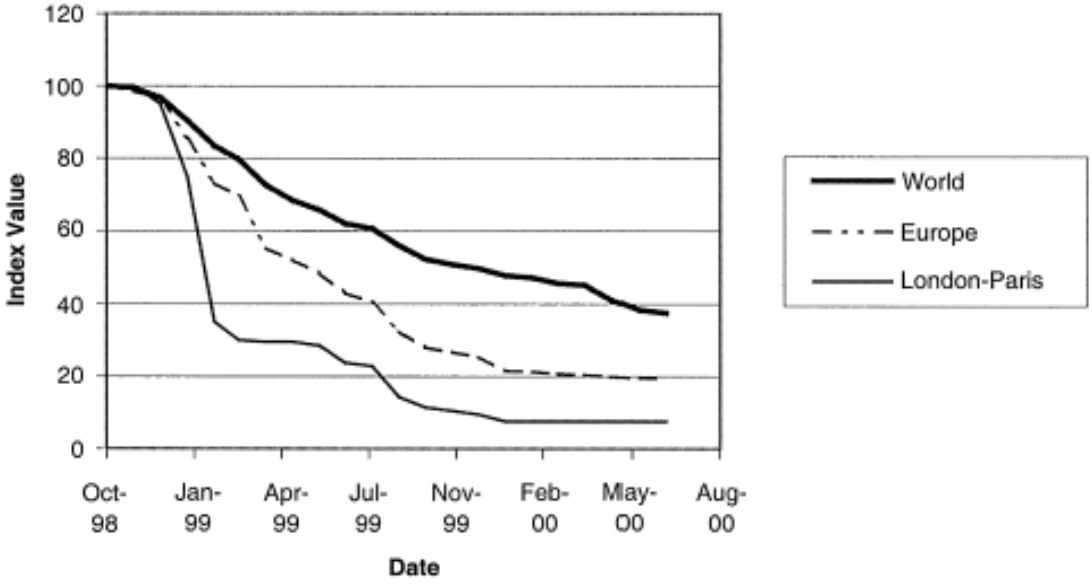


Figure 1-1: Band-X Bandwidth Price Indices
Source: (Galbi, 2001)

1.3 Bandwidth Contracts: The Buyer's Problem

The price for a bandwidth contract at a point in time depends on two parameters: size and duration. Size is the amount of bandwidth that is being purchased, and duration is the length of the contract. For buyers of bandwidth, contracts for large bandwidth may be attractive due to size discounts. Similarly, long-term contracts may be attractive

because of duration discounts. On the other hand, decreasing bandwidth prices over time and demand uncertainties may favor signing short-term contracts. Hence, managers deciding on which contracts to purchase to cover a given planning horizon have to decide on (1) the number of contracts to purchase, (2) the bandwidth size for each contract, and (3) the duration for each contract, with the objective of minimizing total cost.

There is a significant amount of research on problems faced by bandwidth providers under conditions of uncertain demand and price. However, problems faced by bandwidth buyers in contracting for bandwidth are relatively under researched. This problem from a buyer's perspective can be quite complex. For example, if the decision maker is faced with a 36 months planning horizon, and a purchase decision can be made once during each month, then there are 36 decision points for which the decision maker must decide whether to purchase, and if so, the size and duration of each contract to purchase to minimize the total cost over the entire planning horizon.

This research analyzes bandwidth contracting decisions from a buyer's perspective under conditions of discounts for larger bandwidth and longer term contracts and declining prices over time. The parameters of interest are the number of contracts, and the size and duration of each contract. This problem will be solved using mathematical programming and evolutionary algorithms and the solutions will be compared. Two models will be examined. In the first model, demand and price are treated as deterministic. In the second model, demand and price are treated as random parameters that change in the mean over the planning horizon. Both models will be solved using an evolutionary algorithm specifically designed for the bandwidth contract

problem. For the stochastic model a simulation will be used in conjunction with the evolutionary algorithm.

The remainder of this dissertation is organized as follows. In Chapter 2 the mathematical formulation and assumptions for the bandwidth contract problem are outlined and discussed in detail, as well as a discussion of the techniques which will be employed to evaluate the performance of the two evolutionary algorithm implementations. Chapter 3 gives a general overview of evolutionary algorithms and a discussion of how the deterministic evolutionary algorithm (DEA) and stochastic evolutionary algorithm (SEA) (with simulation) will be implemented in order to solve the bandwidth contract problem (BCP). Chapter 4 gives a detailed description of the DBCP-EA and the solution quality. Chapter 5 is discusses the implementation of the SBCP-EA and the results obtained. Chapter 6 contains a summary of this work and final comments, conclusions, and possible areas for future research.

CHAPTER 2 THE BANDWIDTH CONTRACT PROBLEM (BCP)

There is increasing recognition that the rapid proliferation of information networks will significantly impact business practices. A significant body of literature on a variety of telecommunications related topics including economic and policy aspects of networking (Econmides, 1996), pricing (Cochi, Shenkar, Estrin, and Zhang, 1993), network design (Balakrishnan, Magnanti, and Mirchandani, 1998), and a variety of other issues (Shapiro, and Varian, 1999) exists.

The problem of installing or expanding telecommunications capacity under conditions of uncertain demand from a telecommunication service provider's perspective has been studied by many researchers. Multi-period expansion of telecommunications networks has been studied by researchers who typically examined tradeoffs between waiting to invest, revenue opportunities, and maintenance costs for older equipment. Early research by Zadeh (1974) and Minoux (1987) emphasize the complexity and dynamic nature of the problem. Some researchers such as Balakrishnan, Magnanti and Wong (1995) focus on capacity expansion in the context of a portion of a telecommunications network, while others (Chang, and Gavish, 1995; Gavish, 1992) examine the capacity expansion problems for the entire backbone network. This literature basically examines how much capacity to add to each telecommunications link in a network during multiple periods of time. Researchers have also explored

alternative solution techniques for mathematical programming models in telecommunications network design (Premkumar, and Chu, 2000). Mathematical models have also been developed to study telecommunications pricing decisions of suppliers (Brown, and Norgaard, 1992). Other researchers (Chen, Hassin, and Tzur, 2002) analyzed the problem of allocating fixed capacity bandwidth and storage among a number of requests to maximize profits of the seller. Keon and Anandalingam, (2003) present a model for optimally pricing different classes of telecommunications service (voice, data, etc.) from the telecommunications provider's perspective. This dissertation, though related, examines a different problem. The focus here is on acquiring capacity from a buyer's perspective, instead of the telecommunications provider or seller's perspective.

This research is also related to research on future manufacturing capacity procurements under conditions of increasing costs and uncertain demands. Burnetas and Gilbert (2001) examine a scenario where the decision maker trades off waiting for improved demand information with increasing costs of waiting and uses dynamic programming and numerical analysis to decide on the capacity for each period.

Khouja and Kumar (2004) also addressed the bandwidth contract problem from a purchaser's perspective. However, in Khouja and Kumar's model overlapping contracts were not allowed, therefore, a single contract covers the demand for any given period. Khouja and Kumar showed that a fixed contract length works as well as variable contract lengths.

2.1 Problem Formulation

For this model, the following assumptions are made:

1. Demand is deterministic and increasing over time,
2. Prices are deterministic and decreasing over time,
3. Discounts with respect to contract size and duration are offered,
4. Overlapping contracts are allowed,
5. Contract durations must be a multiple of τ, where τ is the minimum contract duration, and
6. No shortages are allowed.

Notation used in formulating the model is as follows:

T = Length of planning horizon,

τ = Minimum duration of a contract, e.g., 1 month,

N = Number of contract periods in the planning horizon, $N = T/\tau$,

i = Beginning period index for a contract, $i = 1, 2 \ldots N$,

j = Ending period index for a contract, $j = 2 \ldots N+1$,

D_i = Demand for bandwidth in period i,

$$x_{i,j} = \begin{cases} 1 \text{ if a contract beginning at period } i \text{ and ending at the beginning of period } j \text{ is purchased,} \\ 0 \text{ otherwise,} \end{cases}$$

$P_{i,j}$ = Price per Mbps per time unit (i.e., τ) for a contract beginning in period i and ending at the beginning of period j; a function of size (Q) and duration (L),

$Q_{i,j}$ = Size of a contract that begins in period i and ends at the beginning of period j, and

$L_{i,j} = j - i$, the duration of a contract that begins in period i and ends at the beginning of period j.

The mathematical formulation of the cost minimization problem over the planning horizon is given by:

$$Min\ TC = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} x_{i,j} Q_{i,j} P_{i,j} (j - i). \quad (1)$$

This problem is subject to N constraints, ensuring that purchased bandwidth is sufficient to meet demand in every period.

$$\sum_{k=1}^{i} \sum_{j=i+1}^{N+1} Q_{k,j} \geq D_i \qquad i = 1, 2 \ldots N. \quad (2)$$

In other words, the sum of bandwidth sizes for all contracts active during period i must be greater than or equal to the maximum demand for bandwidth in period i.

It is assumed that the bandwidth demand will increase. For the deterministic model, demand, growing at a constant rate, is modeled by

$$r_D = 1 + g_D, \quad (3)$$

where g_D is the demand growth rate. Therefore, bandwidth demand at time t is given by

$$D_t = a_D\ r_D^t, \quad (4)$$

where a_D is the initial demand at the beginning of the planning horizon.

For the deterministic model, the price of bandwidth is assumed to be decreasing at a constant rate according to

$$r_P = 1 - g_P, \quad (5)$$

where g_P is the price decrease rate. The unit price for bandwidth per unit time at time t can be written as:

$$P_t = a_P \, r_P^t, \tag{6}$$

where a_P is the initial bandwidth price at the beginning of the planning horizon. To solve this problem bandwidth pricing parameters need to be computed by developing an empirical price function. Historical data is analyzed to determine the price function at time t, P_t. Bandwidth price at the beginning of the planning horizon, or any point in time is a function of contract size and duration. Therefore, at the beginning of the planning horizon

$$a_p = f(Q, L). \tag{7}$$

Substituting into (6) gives

$$P_t = f(Q, L) \, r_p^t. \tag{8}$$

Since all contracts start at the beginning of a period and end at the beginning of a period, the price per Mbps per unit time for contract beginning in period i and ending at the beginning of period j is

$$P_{i,j} = f(Q_{i,j}, j-i) \, r_p^{(i-1)\tau} = a_P \, r_p^{(i-1)\tau}. \tag{9}$$

Substituting $a_P \, r_P^{(i-1)\tau}$ from Equation (9) for $P_{i,j}$ in Equation (1) gives the following bandwidth acquisition problem

$$Min \; TC = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} x_{i,j} \, Q_{i,j} \, a_P \, r_P^{(i-1)\tau} (j-i) \tag{10}$$

subject to Equation (2).

When τ is equal to one, it can be removed from Equation (10). The equations that follow assume that τ is equal to one.

2.2 Illustrative Example: N = 5

Consider a problem with a planning horizon of five months, with possible contracts having a minimum length of one month, i.e., $\tau = 1$. Therefore, N = 5, $i = 1, 2 \ldots 5$, and $j = 2, 3 \ldots 6$.

Table 2-1 shows the total number of possible contracts in a problem with $N = 5$ decision points; note the format adopted for the numbering of each contract. Each column represents a unique contract which could be purchased. In Table 2-1, the white cells containing "# ≥ 0" indicate contracts that could potentially be active and the period in which they can be purchased. The white cells, if any, below a first white cell for a purchased contract will also contain the bandwidth size of that contract. In other words, a contract can only be purchased in the period in which its first white cell appears, and if a contract is purchased the bandwidth size for that contract is recorded in all the white cells for that contract. Note that the duration of the contract is indicated by the number of white cells following the initial purchase of the contract. The grey cells indicate the periods when any given contract cannot be active. All white cells for any contract that is not purchased will contain a 0. Table 2-2 illustrates how quickly the BCP can become intractable. Even small increases in the size of the planning horizon (i.e., *N*) will dramatically increase the number of available contracts and, subsequently, exponentially increase the solution search space.

Table 2-1: Contract Period Coverage Matrix for $N = 5$ Periods

		Data Structure Index Number															
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	
Contract Number		Contract [01, 02]	Contract [01, 03]	Contract [01, 04]	Contract [01, 05]	Contract [01, 06]	Contract [02, 03]	Contract [02, 04]	Contract [02, 05]	Contract [02, 06]	Contract [03, 04]	Contract [03, 05]	Contract [03, 06]	Contract [04, 05]	Contract [04, 06]	Contract [05, 06]	**Constraint Minimum Required Bandwidth**
Period	1	≥ 0	≥ 0	≥ 0	≥ 0	≥ 0											$\geq D_1$
	2		●	●	●	●	≥ 0	≥ 0	≥ 0	≥ 0							$\geq D_2$
	3			●	●	●		●	●	●	≥ 0	≥ 0	≥ 0				$\geq D_3$
	4				●	●			●	●		●	●	≥ 0	≥ 0		$\geq D_4$
	5					●				●			●		●	≥ 0	$\geq D_5$

Table 2-2: Number of Possible Contracts for Different Values of *N*

Number of Decision Points (*N*)	**Number of Possible Contracts (i.e., 1/2 *N* (*N*+1))**
2	3
3	6
4	10
5	15
6	21
: :	: :
12	78
: :	: :
18	171
: :	: :
24	300

2.3 Bandwidth Contract Purchasing Problem: Formulations of the Problem

To solve this problem, a mixed integer formulation is used. The mixed integer programming technique uses Equations (1) and (2), and requires the inclusion of a binary variable $x_{i,j}$ to indicate whether or not a given contract is active, i.e., purchased (Williams, 2005). If $x_{i,j} = 1$, then a contract beginning at period i and ending at the beginning of period j is purchased, else if $x_{i,j} = 0$, then no such contract has been purchased. Therefore, using Equations (1) and (2), the following minimization of total cost (*MinTC*) formula is derived:

$$\begin{aligned} MinTC = {} & x_{1,2}Q_{1,2}P_{1,2}(2-1) + x_{1,3}Q_{1,3}P_{1,3}(3-1) + x_{1,4}Q_{1,4}P_{1,4}(4-1) + \\ & x_{1,5}Q_{1,5}P_{1,5}(5-1) + x_{1,6}Q_{1,6}P_{1,6}(6-1) + x_{2,3}Q_{2,3}P_{2,3}(3-2) + \\ & x_{2,4}Q_{2,4}P_{2,4}(4-2) + x_{2,5}Q_{2,5}P_{2,5}(5-2) + x_{2,6}Q_{2,6}P_{2,6}(6-2) + \\ & x_{3,4}Q_{3,4}P_{3,4}(4-3) + x_{3,5}Q_{3,5}P_{3,5}(5-3) + x_{3,6}Q_{3,6}P_{3,6}(6-3) + \\ & x_{4,5}Q_{4,5}P_{4,5}(5-4) + x_{4,6}Q_{4,6}P_{4,6}(6-4) + x_{5,6}Q_{5,6}P_{5,6}(6-5) \end{aligned} \tag{11}$$

subject to:

$$x_{1,2}Q_{1,2} + x_{1,3}Q_{1,3} + x_{1,4}Q_{1,4} + x_{1,5}Q_{1,5} + x_{1,6}Q_{1,6} \geq D_1 \tag{12}$$

$$x_{1,3}Q_{1,3} + x_{1,4}Q_{1,4} + x_{1,5}Q_{1,5} + x_{1,6}Q_{1,6} + x_{2,3}Q_{2,3} + x_{2,4}Q_{2,4} + x_{2,5}Q_{2,5} + x_{2,6}Q_{2,6} \geq D_2 \quad (13)$$

$$x_{1,4}Q_{1,4} + x_{1,5}Q_{1,5} + x_{1,6}Q_{1,6} + x_{2,4}Q_{2,4} + x_{2,5}Q_{2,5} + x_{2,6}Q_{2,6} + x_{3,4}Q_{3,4} + x_{3,5}Q_{3,5} + x_{3,6}Q_{3,6} \geq D_3 \quad (14)$$

$$x_{1,5}Q_{1,5} + x_{1,6}Q_{1,6} + x_{2,5}Q_{2,5} + x_{2,6}Q_{2,6} + x_{3,5}Q_{3,5} + x_{3,6}Q_{3,6} + x_{4,5}Q_{4,5} + x_{4,6}Q_{4,6} \geq D_4 \quad (15)$$

$$x_{1,6}Q_{1,6} + x_{2,6}Q_{2,6} + x_{3,6}Q_{3,6} + x_{4,6}Q_{4,6} + x_{5,6}Q_{5,6} \geq D_5 \quad (16)$$

Equation (12) can be rewritten as.

$$\begin{aligned} Min\ TC = {} & r_P^0 (x_{1,2}Q_{1,2}a_{P_{12}} + 2x_{1,3}Q_{1,3}a_{P_{1,3}} + 3x_{1,4}Q_{1,4}a_{P_{1,4}} + 4x_{1,5}Q_{1,5}a_{P_{1,5}} + \\ & 5x_{1,6}Q_{1,6}a_{P_{1,6}}) + \\ & r_P^1\left(x_{2,3}Q_{2,3}a_{P_{2,3}} + 2x_{2,4}Q_{2,4}a_{P_{2,4}} + 3x_{2,5}Q_{2,5}a_{P_{2,5}} + 4x_{2,6}Q_{2,6}a_{P_{2,6}}\right) + \\ & r_P^2\left(x_{3,4}Q_{3,4}a_{P_{3,4}} + 2x_{3,5}Q_{3,5}a_{P_{3,5}} + 3x_{3,6}Q_{3,6}a_{P_{3,6}}\right) + \\ & r_P^3\left(x_{4,5}Q_{4,5}a_{P_{4,5}} + 2x_{4,6}Q_{4,6}a_{P_{4,6}}\right) + \\ & r_P^4\left(x_{5,6}Q_{5,6}a_{P_{5,6}}\right) \end{aligned}, \quad (17)$$

which is the same as

$$\begin{aligned} Min\ TC = {} & \sum_{j=2}^{N+1}\left((j-1)x_{1,j}Q_{1,j}a_{P_{1,j}}r_{P_{1,j}}^0\right) + \sum_{j=3}^{N+1}\left((j-2)x_{2,j}Q_{2,j}a_{P_{2,j}}r_{P_{2,j}}^1\right) + \\ & \sum_{j=4}^{N+1}\left((j-3)x_{3,j}Q_{3,j}a_{P_{3,j}}r_{P_{3,j}}^2\right) + \sum_{j=5}^{N+1}\left((j-4)x_{4,j}Q_{4,j}a_{P_{4,j}}r_{P_{4,j}}^3\right) + \\ & \sum_{j=6}^{N+1}\left((j-5)x_{5,j}Q_{5,j}a_{P_{5,j}}r_{P_{5,j}}^4\right) \end{aligned} \quad (18)$$

The pricing data given in Table 1-2 is used to estimate the price of bandwidth as a function of size, and duration. Table 1-2 shows contract costs associated with varying bandwidth sizes for durations of 1, 2, and 3 years.

To fit a regression function, the annual price per Mbps data recorded in Table 1-2 is converted to the price per Mbps per month for each combination of Size and Duration listed in Table 1-2. The price per Mbps per month is shown in Table 2-3. Each

observation in Table 2-4 shows the price per Mbps per month with the inverse of size $\left(\frac{1}{Q}\right)$ and duration $\left(\frac{1}{L}\right)$ to reflect the negative relationship price with size and duration. The resulting regression equation, fitted using SPSS, accounted for 98.8% of the variance in the price data. Tables 2-5 show the regression results from the data in Table 2-4.

Table 2-3: Price per Mbps per Month

Contract Duration (L) (Years)	Contract Bandwidth Size (Mbps)	Annual Price/Mbps*	Price/Mbps/Month
1	6	$14,310	$198.7500
1	9	$15,429	$142.8611
1	12	$16,548	$114.9167
1	15	$17,667	$98.1500
1	18	$18,786	$86.9722
1	21	$19,905	$78.9881
1	45	$25,500	$47.2222
2	6	$13,595	$188.8194
2	9	$14,658	$135.7222
2	12	$15,721	$109.1736
2	15	$16,784	$93.2444
2	18	$17,847	$82.6250
2	21	$18,910	$75.0397
2	45	$24,225	$44.8611
3	6	$12,164	$168.9444
3	9	$13,115	$121.4352
3	12	$14,066	$97.6806
3	15	$15,017	$83.4278
3	18	$15,968	$73.9259
3	21	$16,919	$67.1389
3	45	$21,675	$40.1389

*Based on data from Table 1-2

Table 2-4: Price per Mbps per Month (Formulation #1)

Y	**X1**	**X2**
Price/Mbps/Month	1/Size (i.e., 1/Q)	1/Duration (i.e., 1/L)
198.7500	0.1667	0.0833
142.8611	0.1111	0.0833
114.9167	0.0833	0.0833
98.1500	0.0667	0.0833
86.9722	0.0556	0.0833
78.9881	0.0476	0.0833
47.2222	0.0222	0.0833
188.8194	0.1667	0.0417
135.7222	0.1111	0.0417
109.1736	0.0833	0.0417
93.2444	0.0667	0.0417
82.6250	0.0556	0.0417
75.0397	0.0476	0.0417
44.8611	0.0222	0.0417
168.9444	0.1667	0.0278
121.4352	0.1111	0.0278
97.6806	0.0833	0.0278
83.4278	0.0667	0.0278
73.9259	0.0556	0.0278
67.1389	0.0476	0.0278
40.1389	0.0222	0.0278

Table 2-5: Regression Model Output (Formulation #1)

Regression Statistics	
Multiple R	0.994
R Square	0.989
Adjusted R Square	**0.988**
Standard Error	4.911
Observations	21

		Coefficients	*Standard Error*	*t Stat*	*P-value*
β_1 =	Intercept	**13.116**	3.191	4.110	0.001
β_2 =	1/Size (i.e., 1/Q)	**962.927**	24.266	39.682	0.000
β_3 =	1/Duration (i.e., 1/L)	**258.526**	45.446	5.689	0.000

From Table 2-5 we find the fitted regression equation, which is

$$a_P = 13.116 + \frac{962.927}{Q} + \frac{258.526}{L}, \tag{19}$$

where a_p is the initial price per Mbps per month at the beginning of the planning horizon. Equation (19) is the price function at a point in time. To develop a longitudinal price function to estimate r_p historical data over time will need to be analyzed.

Substituting from Equation (19) into Equation (10) gives the total cost (TC_I) as

$$TC_I = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(x_{i,j}\ Q_{i,j}(j-i)\ r_p^{(i-1)} \left(\beta_1 + \frac{\beta_2}{Q_{i,j}} + \frac{\beta_3}{(j-i)} \right) \right). \tag{20}$$

TC in Equation (20) can be divided into three parts, TC_1, TC_2, and TC_3.

$$TC_1 = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(x_{i,j} Q_{i,j} (j-i)\, r_p^{(i-1)} \beta_1 \right), \tag{21}$$

$$TC_2 = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(x_{i,j}\ Q_{i,j}(j-i)\ r_p^{(i-1)} \frac{\beta_2}{Q_{i,j}} \right), \text{ and} \tag{22}$$

$$TC_3 = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(x_{i,j}\ Q_{i,j}(j-i)\ r_p^{(i-1)} \frac{\beta_3}{(j-i)} \right). \tag{23}$$

The $x_{i,j}$ binary variable is redundant in Equations (21) and (23) since if the contract is purchased the size will be greater than 0, and if the contract is not purchased the size will be 0. Therefore, Equations (23) and (25) can be rewritten without $x_{i,j}$. In addition, Equations (22) and (23) are further simplified. Therefore, Equations (21), (22), and (23) become

$$TC_1 = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(Q_{i,j} (j-i)\, r_p^{(i-1)} \beta_1 \right) \tag{24}$$

$$TC_2 = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(x_{i,j}\ (j-i)\ r_p^{(i-1)} \beta_2 \right), \text{ and} \tag{25}$$

$$TC_3 = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(Q_{i,j}\ \ r_p^{(i-1)} \beta_3 \right). \tag{26}$$

This problem can also be formulated as a non-linear programming problem with linear constraints. This is done by raising the inverse *Size* variable by a value lambda prior to performing regression. Tests showed that a lambda value just less than one fit the data best. Therefore, a lambda of 0.999 was selected to reduce any inflation in the cost formulas. The advantage of this second formulation is that it reduces the complexity of the problem by avoiding mixed integer programming. To illustrate this second problem formulation, the data in Table 1-2 is used again. Notice that Table 2-6 is identical to Table 2-4 with the exception that the values in column labeled "X1" are now equal to $\frac{1}{\text{Size}^{0.999}}$ $\left(\text{i.e., } \frac{1}{Q^{0.999}}\right)$. Based on the data in Table 2-6, the resulting regression equation, fitted using SPSS, accounts for 98.8% of the variance in the price data. Table 2-7 shows the new regression results. This formulation is simpler to solve than the first formulation and just as accurate.

Table 2-6: Price per Mbps per Month (Formulation #2)

Y	**X1**	**X2**
Price/Mbps/Month	$1/Size^{0.999}$ (i.e., $1/Q^{0.999}$)	1/Duration (i.e., 1/L)
$198.7500	0.1670	0.0833
$142.8611	0.1114	0.0833
$114.9167	0.0835	0.0833
$98.1500	0.0668	0.0833
$86.9722	0.0557	0.0833
$78.9881	0.0478	0.0833
$47.2222	0.0223	0.0833
$188.8194	0.1670	0.0417
$135.7222	0.1114	0.0417
$109.1736	0.0835	0.0417
$93.2444	0.0668	0.0417
$82.6250	0.0557	0.0417
$75.0397	0.0478	0.0417
$44.8611	0.0223	0.0417
$168.9444	0.1670	0.0278
$121.4352	0.1114	0.0278
$97.6806	0.0835	0.0278
$83.4278	0.0668	0.0278
$73.9259	0.0557	0.0278
$67.1389	0.0478	0.0278
$40.1389	0.0223	0.0278

Table 2-7: Regression Model Output (Formulation #2)

Regression Statistics	
Multiple R	0.994
R Square	0.989
Adjusted R Square	**0.988**
Standard Error	4.911
Observations	21.000

		Coefficients	*Standard Error*	*t Stat*	*P-value*
β_1 =	Intercept	**13.051**	3.191	4.090	0.001
β_2 =	1/Size^0.999 (i.e., 1/Q^0.999)	**961.753**	24.239	39.678	0.000
β_3 =	1/Duration (L)	**258.169**	45.399	5.687	0.000

From Table 2-7, the fitted regression equation is

$$a_P = 13.051 + \frac{961.753}{Q^{0.999}} + \frac{258.169}{L}. \tag{27}$$

Substituting from Equation (27) into Equation (10), and removing the binary variable, $x_{i,j}$, gives the non-linear total cost (TC_C) as

$$TC_C = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(Q_{i,j} \left(j-i\right) r_p^{(i-1)} \left(\beta_1 + \frac{\beta_2}{Q_{i,j}^{0.999}} + \frac{\beta_3}{\left(j-i\right)} \right) \right). \tag{28}$$

TC_C in Equation (28) can be divided into three parts, $TC_{C1}, TC_{C2},$ and TC_{C3}.

$$TC_{C1} = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(Q_{i,j} \left(j-i\right) r_p^{(i-1)} \beta_1 \right), \tag{29}$$

$$TC_{C2} = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(Q_{i,j} \left(j-i\right) r_p^{(i-1)} \frac{\beta_2}{Q_{i,j}^{0.999}} \right), \text{ and} \tag{30}$$

$$TC_{C3} = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(Q_{i,j} \left(j-i\right) r_p^{(i-1)} \frac{\beta_3}{\left(j-i\right)} \right). \tag{31}$$

Equations (30) and (31) reduce to

$$TC_{C2} = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(Q_{i,j}^{0.001} \left(j-i\right) r_p^{(i-1)} \beta_2 \right), \tag{32}$$

and

$$TC_{C3} = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left(Q_{i,j} \, r_p^{(i-1)} \beta_3 \right). \tag{33}$$

The binary variable $x_{i,j}$ was needed in the first formulation because of Equation (25). In the second formulation, we take advantage of the fact that if a contract does not exist its size is zero $\left(\text{i.e., } Q_{i,j} = 0\right)$, and since Equations (29), (32), and (33) all contain $Q_{i,j}$, the total costs ($TC_{C1}, TC_{C2},$ and TC_{C3}) will only contain the sum totals of valid

contracts. If the binary variable were not used in Equation (25) it would reduce to $TC_2 = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left((j-i)\, r_p^{(i-1)} \beta_2 \right)$, which is incorrect because there is no way to indicate whether a contract has been purchased or not. This formulation would sum all possible contracts, not just those contracts that have been purchased.

Two ways to formulate this problem have been illustrated: (1) mixed integer linear programming, and (2) non-linear programming. There are many software packages which might be used to solve the mixed integer linear programming problem; for example, CPLEX (CPLEX, 1995). To solve the second programming formulation, non-linear objective function with linear constraints, a Lagrangian function could be used.

The bandwidth contract purchasing problem presented in the rest of this dissertation will be solved using Formulation #1, which uses mixed integer linear programming (MILP), since it is easier to identify optimal solutions when solving a linear problem as opposed to a non-linear problem. The solution and computational time of the evolutionary algorithms (a meta-heuristic, which will be discussed in detail in Chapter 3), are then compared and evaluated based on the solutions obtained by the mixed integer program (Williams, 2005) solved using CPLEX.

Chapter 3 includes a general overview of evolutionary algorithms and a discussion of how the proposed evolutionary algorithm for the bandwidth contract problem, for both a deterministic model and a stochastic model, will be implemented.

CHAPTER 3 EVOLUTIONARY ALGORITHMS: A POPULATION BASED HEURISTIC SEARCH ALGORITHMS

As stated earlier, Khouja and Kumar (2004) also addressed the bandwidth contract problem. The research proposed here is different in that it will use evolutionary algorithms to solve the problem. An evolutionary algorithm provides a framework that is more flexible than the model used by Khouja and Kumar (2004). Evolutionary algorithms can deal with many functional forms of price and demand changes, whereas Khouja and Kumar's formulation can only handle fixed percentages of increase and decrease in price and demand. In addition, evolutionary algorithms do not require that functions be continuous, i.e., step functions can easily be used.

The purpose of this chapter is to provide a brief overview of evolutionary algorithms followed by a discussion of how evolutionary algorithms will be implemented in solving the bandwidth contract problem, hereafter referred to as BCP. The performance of the deterministic implementation of the evolutionary algorithm will be evaluated through a comparison with results obtained through mathematical programming methods. Finally, the chapter will conclude with a discussion of a stochastic implementation of an evolutionary algorithm with a simulation for the BCP.

3.1 Evolutionary Algorithms

Evolutionary algorithms, hereafter EAs, are a problem solving technique, known as a meta-heuristics. EAs use the concept of evolution and hereditary to produce quality solutions to complex problems that have large search spaces and are, therefore, difficult

to solve. EAs have been used to solve many types of complex problems (Aytug, Khouja and Vergara, 2003). A well designed EA allows for the efficient and effective exploration and exploitation of a problem's solution space in an effort to identify the global optimal, or a near optimal, solution.

EAs create and manipulate a group of possible solutions referred to as a population. Each possible solution within the population is called a chromosome. The population undergoes change throughout the run of the EA thereby evolving the population of chromosomes toward a best solution. Within the EA, the population loops through a series of processes numerous times; a complete execution of all procedures is referred to as a generation. Hence, throughout the run of the EA, the population will cycle through many generations as the chromosomes within each subsequent population change and evolve, until, it is hoped, a good solution emerges. The processes encountered within each generation include an evaluation process, an alteration process, and a selection process. These processes may occur in various orders; however each is required at each generation (Michalewicz, 1996).

The evaluation process uses an evaluation function that assesses the relative fitness, or "goodness," of each chromosome within the population at each generation. In addition, at each generation a number of chromosomes are subjected to some form of perturbation. These perturbations constitute the alteration process. These changes are manifested through the use of specific algorithmic operations, called genetic operators. Genetic operators can be either mutation operators, which introduce small changes within a single chromosome, or crossover operators, which cut and paste different parts from two or more chromosomes in order to create two or more new chromosomes. The

probability of a chromosome experiencing some form of perturbation within any given generation is subject to predefined parameters for each of the operators. Therefore, the probability of mutation and the probability of crossover will determine how many chromosomes are altered. Through the alteration process, some, or all, of the chromosomes within a generation's population are altered. These altered chromosomes are often called offspring, and the chromosomes that are used to create the offspring are often called parents. Offspring might replace their parents, or they might simply be added to the existing population, thereby creating a large pool from which the next generation can be created. Finally, a selection process used within an EA provides a procedure for selecting parents so that offspring can be created, and is also used to promote the survival of the best chromosomes from one generation to the next (i.e., generational policy). When properly implemented the use of selective pressure encourages the population to converge to a quality solution.

Each EA is unique in its design with regard to several important elements depending on the problem domain and the programmer's preferences. However, regardless of the differences, all EAs attempt to evolve the chromosomes within the population through the use of genetic operators and selective pressures to converge to a good solution to complex problems. So, while each manifestation of an EA is unique; it is important to begin a discussion of EA by describing a few relevant aspects that are common to most, if not all, EAs. These elements include genetic representation, method for creating the initial population, genetic operators, evaluation function, selection method, generational policy, terminating conditions, parameters, and constraint handling techniques.

3.2 Genetic Representation

The genetic representation, or data structure, of an EA specifies how a problem will be represented within the computer. Each problem is unique and the data structure used to solve a problem must be carefully designed to accurately model the problem. The classic representation within an EA is to use binary digits. However, other representations such as integer and floating point have been found to yield better solutions for different problems. The use of an inappropriate coding scheme has been the cause of many GA failures (Reeves, 1997).

3.3 Method for Creating the Initial Population

Often initial populations are generated randomly. For problems with small feasible regions, initialization can incorporate problem-specific knowledge to increase the likelihood of having feasible chromosomes and to generate some good solutions in the initial population.

3.4 Genetic Operators

Genetic operators alter the genetic composition of parent chromosomes during reproduction, thereby creating offspring. Genetic operations can include crossover, mutation, or both. In addition, a given EA might employ a variety of crossover and/or mutation operators. The operators that manipulate the chromosomes must be carefully selected so that alterations performed on the chromosomes are meaningful and promote diversity without unduly introducing infeasible solutions into the population.

Of the two types of genetic operators, mutation changes one chromosome only slightly. Mutation can be as simple as randomly changing the value stored within one gene within a chromosome. The simplest mutation for a binary vector is to flip a bit in a

gene (index) within the chromosome. Other mutation operators include swapping the values between two genes, randomly inserting the value of one gene into another location and shifting, etc. Mutation moves the EA to a different neighborhood of the search space (Vose and Liepins, 1991), and is usually called the 'exploration' operator. Mutation, therefore allows the EA to explore diverse regions in the search space.

The second genetic operator, crossover, is called the focusing operator, enabling the EA to exploit the current neighborhood and is expected to move the EA to a local optimum. Crossover exchanges genetic material between two or more parents. A one-point crossover exchanges all genes to the left of a cut-point whereas a two-point crossover exchanges genes between two cut-points. Cut-points are usually randomly determined. Most disruptive is the uniform crossover, where genes from both parent are randomly swapped.

3.5 Evaluation Function

An evaluation function (a.k.a. fitness value) is used to evaluate the "goodness," or "fitness," of each chromosome with regard to the total cost of the purchasing strategy represented in the data structure of each chromosome. Since each chromosome represents a potential solution to a problem, the evaluation function assigns a real number as a measure of fitness to each chromosome.

3.6 Selection Method

There are many methods to employ selective pressure within an EA. A selection method is a scheme for selecting chromosomes, either for undergoing an alteration process, or for generational policy, which involves selecting chromosome for a new population at the beginning of each new generation. Two popular selection methods are

the roulette wheel and tournament. The roulette wheel gives chromosomes a chance of selection equal to their fitness relative to the population. The higher a chromosome's fitness ratio is with respect to the overall population, the higher the probability that it will be selected. Tournament selection randomly places k chromosomes $(k \geq 2)$ in competition against each other. The winner is the chromosome with the best fitness value. The winner then actively contributes to the next generation, by either participating in the alteration process, or by occupying a place in the next generation's population. An additional method often used is random selection, which selects chromosomes for either alteration, or to enter the next generation's population, in a completely arbitrary manner.

3.7 Generational Policy

A replacement scheme for creating each new generation is called the generational policy. Replacement strategies specify how the next generation is to be created. Often, offspring replace their parents. However, there are many variations to this rule. One commonly applied strategy is the elitist strategy, which always carries at least one copy of the best chromosome to the next generation. A tournament strategy is based on a tournament scheme, where the winner of a contest between two or more chromosomes is copied to the next generation. Another scheme uses each offspring as a starting point for a local search algorithm and accepts the resulting, and improved, solutions as a new offspring, which is then carried over into the next generation.

3.8 Termination Conditions

The termination condition in an EA is essentially the stopping criteria. Running the EA for a predetermined number of generations is the most common criterion, however, time-independent criterion such as population entropy or diversity have also

been used. Entropy is a term borrowed from physics and is generally used to refer to the gradual loss of heat or "cooling" of a system from a chaotic, excited, state to a state of balance and equilibrium. The use of entropy within an EA is modeled as a function that slowly degrades over many generations until its value falls below some given threshold value, at which point the program terminates. Diversity reflects the difference between the chromosomes within a population. Diversity in EAs can be implemented in several ways. One way diversity can be measured is as a pairwise comparison of the values in corresponding indexes between chromosomes. For example, by comparing the commonalities, index by index, among chromosomes the EA could be programmed to terminate when the population reaches some measure of homogeneity. A second way that diversity can be measured is as the difference in the fitness value between chromosomes within a population. For example, a common criterion sometimes used is to monitor the average (or best) fitness value within each successive generation, and when the number of generations where the average (or best) fitness value does not improve is greater than some set threshold value, the program terminates. Another possibility is to set a threshold value, and when the difference between fitness values for the best and worst chromosomes within a population is less than this threshold value the program terminates.

3.9 Computational Parameters

The parameters define the settings used to run an EA. EA parameter selection includes the setting of values for population size, crossover and mutation probabilities rates, and stopping criteria. There is no definitive process for choosing these parameters. The practice is to use parameters based on pilot runs or ad-hoc selection.

3.10 Constraint Handling Techniques

Most problems have constraints. For example, for the bandwidth contract problem there are N constraints, given by Equation (2). These constraints stipulate that shortages are not allowed for any period. Infeasibility can result if a constraint is violated. Some EAs allow for the inclusion of such infeasible solutions; others include repair algorithms to transform them into feasible solutions. Still others simply delete infeasible solutions when they occur (Michalewicz, 1996). It is up to the EA designer to decide which course of action is appropriate. A major advantage of EAs is there ability to solve highly constrained problems (Michalewicz, 1996).

CHAPTER 4 THE DETERMINISTIC BANDWIDTH CONTRACT PROBLEM EVOLUTIONARY ALGORITHM (DBCP-EA)

The proposed EA identifies an optimal or near optimal solution for the DBCP in a relatively short period of time. The DBCP-EA is designed to allow multiple overlapping contracts, and contains a repair function that transforms infeasible solutions into feasible solutions. A feasible solution is a solution where the total purchased bandwidth for all contracts active during each period in a planning horizon is equal to or greater than the demand for each period. In testing the EA, each problem has different parameter settings for each of the four parameters: planning horizon length, temporal discount, price decrease rate, and demand increase rate. Each of these parameters can assume one of three values resulting in $3 \times 3 \times 3 \times 3 = 81$ problems.

Several variable labels are used within this chapter to facilitate the discussion of the various aspects of the DBCP-EA. A listing of these labels and a description of each is presented here.

%_Error: The mean percent error (i.e., the distance from the optimal solution). For the DBCP-EA it is the mean percent error of 5 runs for each problem.

MOM_%_Error: The mean of the mean percent errors (i.e., the average of a group of *%_Error*). This variable is used when discussing observations from aggregated *%_Error* data with respect to the four parameters.

Pop_Size: The number of chromosomes used throughout the run of the DBCP-EA. This value is held constant at 40.

The number of times to run each problem was determined based on preliminary tests run on sample problems. It was found that running each problem for five runs produced a good representation of the variability in DBCP-EA solution quality. The population size (*Pop_Size*) was identified based on an analysis of the tradeoff between population size, where larger populations might potentially achieving better solution quality, versus the computational run time required to solve each problem. A *Pop_Size* of 40 chromosomes was found to perform well with respect to both solution quality and run time. The DBCP-EA uses two crossover operators and one mutation operator. Each operator has an operator rate which is a parameter setting that determines the probability of applying the operator. Therefore, the *One-point Crossover Rate*, the *Uniform Crossover Rate*, and *Mutation Rate*, are associated with the One-point Crossover Operator, the Uniform Crossover Operator, and the Mutation Operator, respectively.

fv: The measure of 'goodness' of a purchasing strategy recorded in index 0 of a chromosome. For the BCP-EA the total cumulative cost of a purchasing strategy is used as the fitness value for each chromosome.

X: A variable that contains the number of potential contracts available over the planning horizon plus one $\left(\textbf{i.e.,}\ X = \left(1/2\,N\,(N+1)\right)+\mathbf{1}\right)$ for each problem. The length of each chromosome is equal to *X*.

Chromosome_Array: A two dimensional array of width *X* and length *Pop_Size*. This array is used to hold the population of chromosomes at the beginning of each generation during the run of a BCP-EA problem.

Chromosome_Array structure		*X*			
		Index 0	Index 1	…	Index (1/2 N (N+1)
Pop_Size	Chromosome 1				
	Chromosome 2				
	:				
	Chromosome (*Pop_Size*)				

Pool_Array: A two dimensional array of width *X* and of varying length. The final length is equal to the *Pop_Size* plus all of the offspring created by the use of the three genetic operators. At the end of each generation, the next generation's population of chromosomes is selected from the *Pool_Array* and copied into the *Chromosome_Array*.

Pool_Array structure		*X*			
		Index 0	Index 1	…	Index (1/2 N (N+1)
Pop_Size + # of Offspring	Chromosome 1				
	Chromosome 2				
	:				
	Chromosome (*Pop_Size + # of Offspring*)				

Select next generation's beginning population

Chromosome_Array structure		*X*			
		Index 0	Index 1	…	Index (1/2 N (N+1)
Pop_Size	Chromosome 1				
	Chromosome 2				
	:				
	Chromosome (*Pop_Size*)				

Parent Chromosomes (*e.g., Parent #1* and *Parent #2*): A one dimensional array of length *X* used during a tournament selection during the use of the crossover and mutation operators. These arrays contain two chromosomes from the population.

Offspring (*e.g., Offspring #1* and *Offspring #2*): A one dimensional array of length *X* used within the crossover and mutation operators. The *Parent Chromosome(s)* is/are altered and the resulting chromosome(s) is/are called *Offspring(s)*. Crossover operators create two *Offspring*; whereas, the mutation operator creates only one.

bws: This the purchased bandwidth size. If a contract is purchased the size of the purchased amount of bandwidth is recorded in the chromosome index corresponding to that contract. Bandwidth sizes are restricted to integer values.

Prob_Purchase: The probability of purchasing a given contract. This value is used when creating the initial population of chromosomes for a DBCP-EA problem.

4.1 DBCP-EA Process Flowchart

Figure 4-1 is an overview of the basic processes within the DBCP-EA. The DBCP-EA pseudo code follows Figure 4-1. Appendix A contains the source code for DBCP-EA. The remaining sections in this chapter discuss the various aspects of the DBCP-EA in detail.

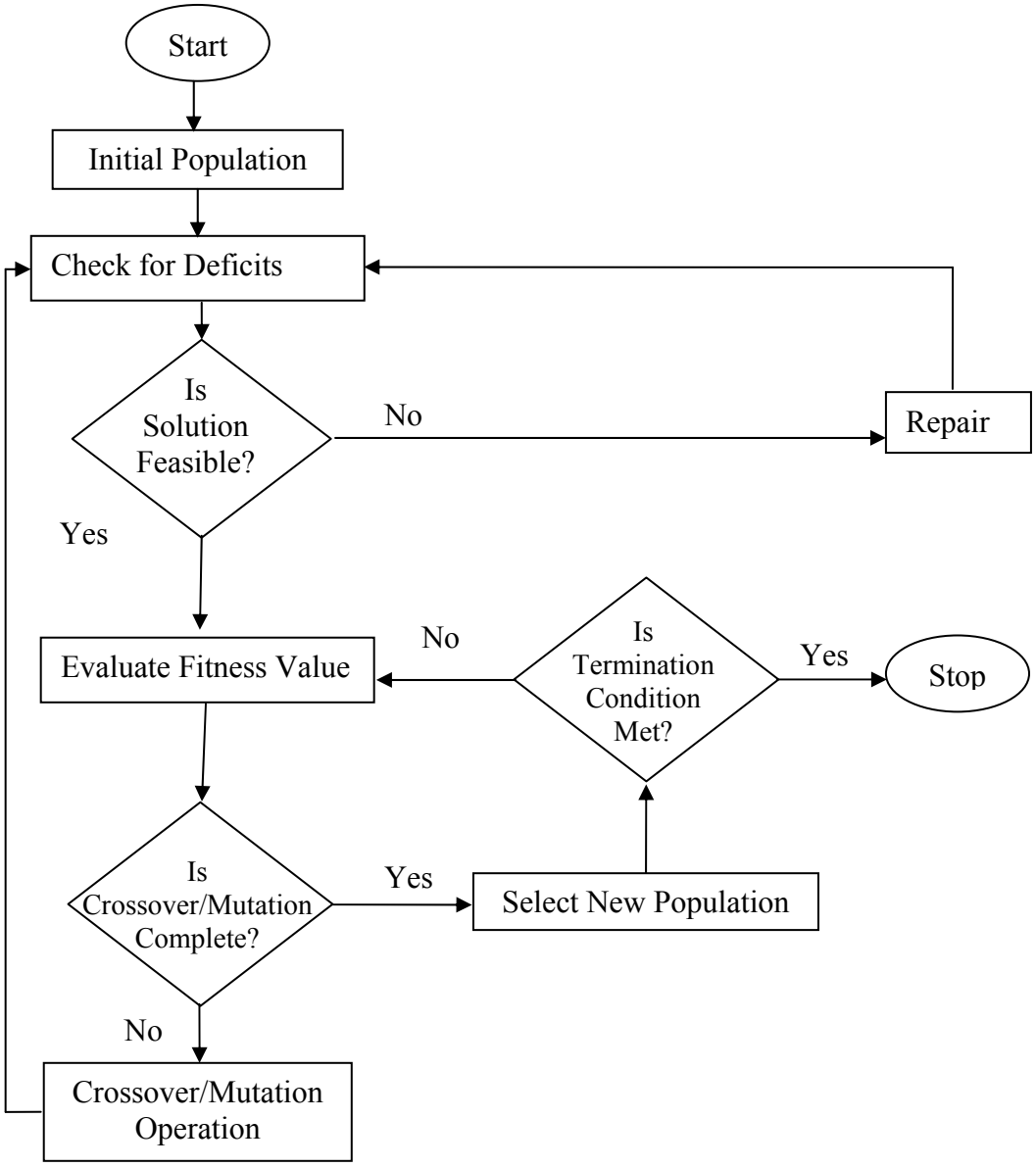


Figure 4-1: DBCP-EA Flowchart

BCP EA Pseudo Code

1. Select *N*. LOOP1 = 0. LOOP2 = 0.
2. Set *Pop_Size*.
3. Set operator parameters: *One-Point Crossover Operator Rate*, *Uniform Crossover Operator Rate*, *and Mutation Operator Rate.* Note: ($0 \leq$ Operator Rate ≤ 1).
4. Set the maximum number of generations to run.

5. Create initial population of chromosomes and place them in the *Chromosome_Array*. Calculate the *fv* for each chromosome and place the value in index 0 of each chromosome vector.
6. LOOP1 = LOOP1 + 1. Set loop to run 3 times, once for each temporal discount scheme.
    a. $s = 1$ for Small temporal discounts
    b. $s = 2$ for Medium temporal discounts
    c. $s = 3$ for Large temporal discounts
7. LOOP2 = LOOP2 + 1. Set loop to run 9 times, once for all possible price and demand combination (i.e., 3 x 3 = 9).
    a. Price decrease rate (p): 0.1, 0.2, and 0.3.
    b. Demand increase rate (d): 0.1, 0.2, and 0.3.
8. Copy generation's chromosomes from the *Chromosome_Array* into the *Pool_Array.*
9. Initiate the one-point crossover operator. Append two offspring to the end of the *Pool_Array* each time the one-point crossover operator is implemented. (See Section 4.1.3.)
10. Initiate the uniform crossover operator. Append two offspring to the end of the *Pool_Array* each time the uniform crossover operator is implemented. (See Section 4.1.3.)
11. Initiate the mutation operator. Append one offspring to the end of the *Pool_Array* each time the mutation operator is implemented. (See Section 4.1.3.)
12. Check all chromosomes in the *Pool_Array* for feasibility. If infeasible, then correct using the repair function. (See Section 4.1.9.)
13. Use the evaluation function to assign a fitness value (i.e., *fv*) to each chromosome in the *Pool_Array*. (See Section 4.1.4.)
14. Select the next generation's chromosomes from the *Pool_Array* and copy them into the *Chromosome_Arrray.* (See Section 4.1.6.)
15. If a termination condition is met, then go to step 16 (See Section 4.1.7). Else, go to step 8.
16. If LOOP2 < 9, then go to step 7. Else, go to step 17.
17. If LOOP1 < 3, then go to step 6. Else, done!

4.1.1 BCP EA Genetic Representation

Chromosomes represent possible solutions, or purchasing strategies, for a given BCP problem. For each BCP problem, the length of the planning horizon is specified.

Therefore, there are N possible contracts that can begin in the first period, and $(N-1)$ possible contracts that begin in the second period, etc. The total number of possible contracts in a planning horizon of size *N* is

$$N+(N-1)+(N-2)+...+1=\frac{1}{2}N(N+1) \tag{34}$$

The BCP EA uses a simple data structure that is designed to be dynamic; allowing the EA to be used for varying values of N without having to modify the EA program code. The data structure is a one dimensional vector of length *X* that contains a feasible solution for a DBCP-EA problem. Figure 4-2 is a graphical representation of the BCP-EA chromosome data structure for a problem with $N=5$ periods, and therefore 15 possible contracts $\left(\text{i.e., } \frac{1}{2}(5(5+1))\right)$.

At the beginning of each generation, a population of chromosomes is recorded in the *Chromosome_Array*. During the run of each generation, the chromosomes from the *Chromosome_Array* are used to create *Offspring* using the three genetic operators. At the end of a generation the *Pool_Array* will contain the *Chromosome_Array* chromosomes plus any *Offspring* that were created. At the end of each generation chromosomes are selected from the *Pool_Array* and copied into the *Chromosome_Array*, thereby generating the beginning population of chromosomes for the next generation.

A problem solution is a sequence of integer numbers recorded in indexes 1 through *X* in a chromosome. Each index represents a unique contract, and each contract contains a *bws*. If the contract has not been purchased its *bws* is zero, else the *bws* is some integer value. Contracts are designated by the period in which they can begin and the period for which the contract expires; contracts begin in period *i* and end at the

beginning of period *j*. Therefore, contracts are identified by their beginning and ending periods (i.e., Contract [*i*, *j*]). For an $N = 5$ problem, index 1 would contain the *bws* for the first available contract in period 1, (i.e., the contract that begins in period 1 and ends at the beginning of period 2; designated as Contract [01, 02]). Index 6 would contain the *bws* for the first contract available in period 2 (i.e., the contact that begins in period 2 and ends at the beginning of period 3; designated as Contract [02, 03]). This indexing scheme concludes with the final contract, associated with index 15 (i.e., the contract that begins in period 5 and ends at the beginning of period 6; designated as Contract [05, 06]). Index 0 is reserved for the chromosome's fitness value (*fv*).

The chromosome's fitness value (i.e. Total Cost)															
Contract Length →		1	2	3	4	5	1	2	3	4	1	2	3	1	2	1
fv	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*	*bws*
00	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*
	→Contract [01, 02]	→Contract [01, 03]	→Contract [01, 04]	→Contract [01, 05]	→Contract [01, 06]	→Contract [02, 03]	→Contract [02, 04]	→Contract [02, 05]	→Contract [02, 06]	→Contract [03, 04]	→Contract [03, 05]	→Contract [03, 06]	→Contract [04, 05]	→Contract [04, 06]	→Contract [05 , 06]

Data structure index number →

Figure 4-2: Chromosome Data Structure for $N = 5$

4.1.2 BCP-EA Method for Creating the Initial Population

The *Pop_Size* of 40 is held constant throughout the run of each problem. The *Chromosome_Array* contains the beginning population of chromosomes. Each row within the *Chromosome_Array* represents a unique chromosome.

For each contract, within each chromosome, a random number (*RandNum*) is generated between 0 and 1. If $RandNum \leq Prob\_Purchase$, then an integer *bws* value is randomly generated. If $RandNum > Prob\_Purchase$, then $bws = 0$. Therefore, a $bws > 0$ indicates that the contract has been purchased. Figure 4-3 shows an example of how a randomly generated chromosome might appear.

Initialize Chromosomes Pseudo Code

1. Select *Pop_Size* (The population size is set to 40 chromosomes).
2. Set value for *Prob_Purchase* ($0 \leq Prob\_Purchase \leq 1$).
3. Set variable $Z = 1$.
4. For each index from 1 to X in chromosome number Z generate a random number. If the random number is greater than the *Prob_Purchase* then enter 0 into the index's data location. Else, generate a *bws* and place that value in the index's data location.
5. $Z = Z + 1$.
6. If $Z < Pop\_Size$, then go to step 2. Else, stop.

4.1.3 Three BCP-EA Genetic Operators

The three genetic operators used in the BCP-EA are: (1) a one-point crossover (see Figure 4-4), (2) a uniform crossover operator (see Figure 4-5), and (3) a mutation operator (see Figure 4-6). These three operators are used by the EA to create additional chromosomes called *Offspring* by altering copies of existing chromosomes. All three operators work with the BCP-EA data structure.

The one-point crossover operator uses two *Parent Chromosomes* to create two *Offspring*. If the two *Parent Chromosomes* were stacked on top of each, then the

chromosome indexes would line up. A cut point between 1 and *X* is randomly generated. All the contract data to the left of the cut point in *Parent #1* are copied into *Offspring #1*; likewise the contract data to the left of the cut point in *Parent #2* are copied into *Offspring #2*. Then the contact data to the right of the cut point in *Parent #1* are copied into *Offspring #2*, and the contract data to the right of the cut point in *Parent #2* are copied into *Offspring #1***.**

One-point Crossover Operator Pseudo Code

1. Set crossover to run (*Pop_Size*/2) times
2. Generate a random number (*RandNum*) between 0 and 1. If *RandNum* < *One-point Crossover Operator Rate*, then go to step 3. Else, go to step 9.
3. Set Z = 1.
4. Randomly generate a number between 1 and *X*. This randomly generated number is the cut point.
5. Randomly generate two numbers between 1 and *Pop_Size* to select two chromosomes from the *Chromosome_Array*. Perform a tournament selection between the two chromosomes. The chromosome with the smaller fitness value (see Section 4.1.4) is the winner and is labeled *Parent #1*.
6. Repeat step 5 to identify *Parent #2*.
7. For each parent, copy all the values from the data locations from index 1 up to the cut point and paste the data into one of the two offspring chromosomes. Next, copy the values in the remaining data locations from both parents into the alternate offspring.
8. Copy the two resulting *Offspring* (*Offspring #1* and *Offspring #2*) into the *Pool_Array*.
9. If Z < (*Pop_Size*/2), then go to step 2. Else, stop.

For the uniform crossover operator, a random number (*RandNum*) is generated. If *RandNum* < *Uniform Crossover Operator Rate* two *Offspring* are created. The procedure for selecting *Parent Chromosomes* is the same as described for the one-point crossover operator.

For each index between 1 and *X* a random binary number (*BinNum*) is generated. If *BinNum* = 0, then the value in the data location for that index in *Parent #1* is copied

into *Offspring #1*, and the value in the same data location in *Parent #2* is copied into *Offspring #2*. If *BinNum* = 1, then the value in that index's data location in *Parent #1* is copied into *Offspring #2*; likewise the value in the same data location in *Parent #2* is copied into *Offspring #1*.

Uniform Crossover Operator Pseudo Code

1. Set crossover to run (*Pop_Size*/2) times.
2. Randomly generate a number (*RandNum*) between 0 and 1. If *RandNum* < *Uniform Crossover Operator Rate*, then go to step 3. Else, go to step 12.
3. Set *Count* = 0 and *Z* = 1.
4. Randomly generate two numbers between 1 and (*Pop_Size*) to select two chromosomes from the *Chromosome_Array*. Perform a tournament selection between the two chromosomes. The chromosome with the smaller fitness value (see Section 4.1.4) is the winner and is labeled *Parent #1*.
5. Repeat step 4 to identify *Parent #2*.
6. *Count* = *Count* + 1.
7. Randomly generate a binary number (*BinNum*). If *BinNum* = 1, then go to step 8. Else, go to step 9.
8. Copy the value in the data location for index *Count* in *Parent #1* into index *Count* in *Offspring #1*. At the same time, copy the value in the data location for index *Count* in *Parent #2* into the corresponding data location of *Offspring #2*.
9. Copy the value in the data location for the index *Count* in *Parent #1* into the same data location in *Offspring #2*. At the same time, copy the value in the data location of index *Count* in *Parent #2* into index *Count* of *Offspring #1*. If *Count* = *X*, then go to step 10. Else, go to step 6.
10. Copy the two *Offspring* into the *Pool_Array*.
11. Z = Z + 1.
12. If Z < (*Pop_Size*/2), then go to step 2. Else, stop.

The mutation operator alters one chromosome to create one *Offspring*. The *Parent Chromosome* is randomly selected from the *Chromosome_Array*. For each index between 1 and *X* in the *Parent Chromosome* a number is randomly generated (*RandNum*). If *RandNum* ≤ *Mutation Operator Rate*, a *bws* is generated and placed in the corresponding data location in the *Offspring*. If *RandNum* > *Mutation Operator Rate*, the

value in the data location for that index in the *Parent Chromosome* is copied into the *Offspring*.

Mutation Operator Pseudo Code

1. Set mutation to run (*Pop_Size*/2) times.
2. Set *Count1* = 0, *Count2* = 1, and *Z* = 1.
3. Randomly generate two numbers between 1 and *Pop_Size* to select two chromosomes from the *Chromosome_Array*. Perform a tournament selection between the two chromosomes. The chromosome with the smaller fitness value (see Section 4.1.4) is the winner, i.e., the *Parent.*
4. *Count1* = *Count1* + 1.
5. Randomly generate a binary number (*BinNum*) between 0 and 1. If *BinNum* > *Mutation Operator Rate*, copy the value in the data location for index *Count1* in the *Parent* chromosome into the *Offspring*. While *Count1* < *X*, go to step 4. Else, go to step 6.
6. Check the *Offspring* for feasibility. If the *Offspring* is infeasible, run the repair function. Calculate present *fv* of the *Offspring*.
7. For each index, if the *bws* > 0, then generate a random number (*BinNum*) between 0 and 1. If *BinNum* > *Mutation Operator Rate*, then generate a *bws*. Subtract the *bws* from the existing *bws* value in the data location for that index. Since, *bws* can't be negative, if the resulting *bws* < 0, then set the *bws* = 0.
8. Check the *Offspring* for feasibility. If the *Offspring* is infeasible, run the repair function (see Section 4.1.9). Recalculate the *fv* of the *Offspring* (see Section 4.1.4). *Count2* = *Count2* + 1. If the new re-calculated *fv* is less than the previous *fv* and *Count2* < 20, then go to step 7. Else, go to step 9.
9. Copy the *Offspring* chromosome into the *Pool_Array*.
10. *Z* = *Z* + 1.
11. If *Z* < (*Pop_Size*/2), then go to step 2. Else, stop.

At the end of each generation the beginning chromosomes and all the *Offspring* created by the three genetic operators are located in the *Pool_Array*. The next generation's population is selected from the *Pool_Array* after each chromosome has been checked, and if necessary repaired, for feasibility.

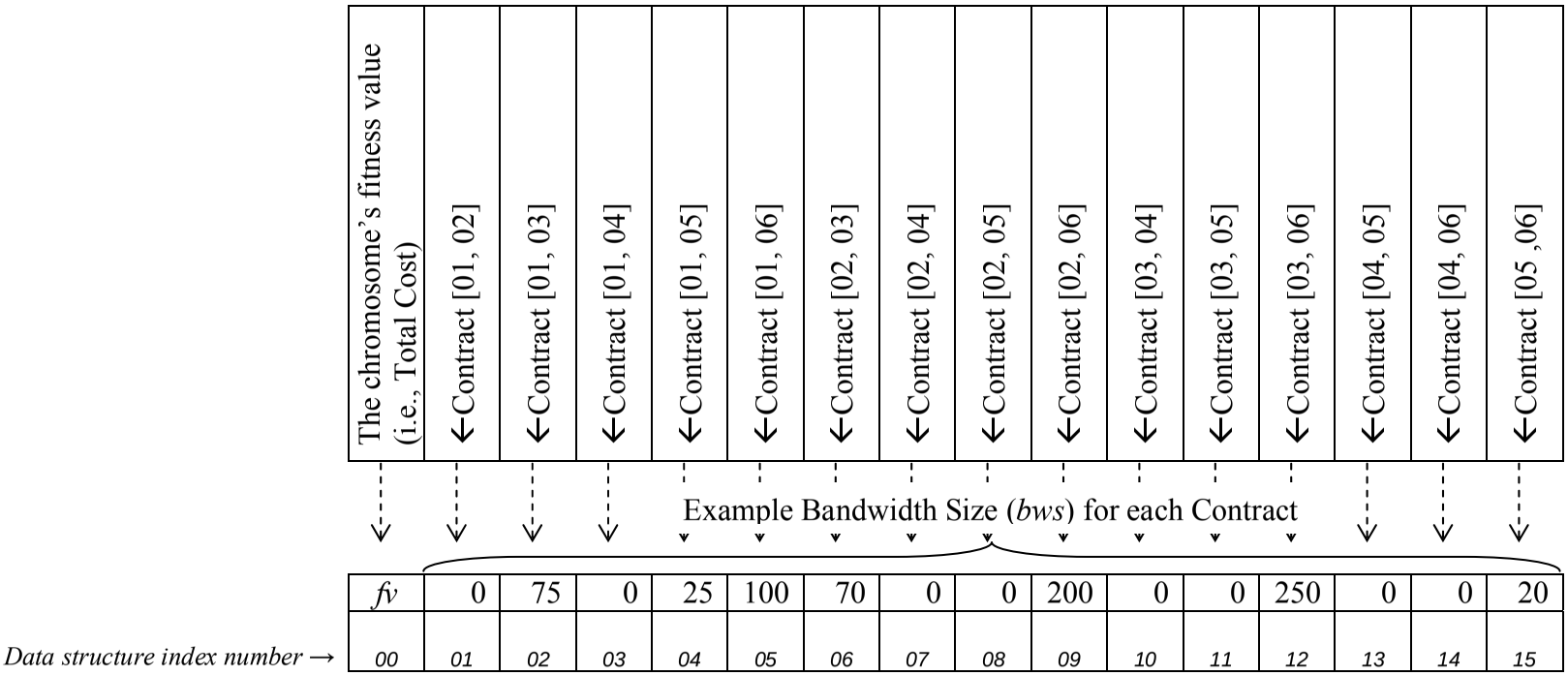


Figure 4-3: Example of a Randomly Generated Initial Chromosome for $N = 5$

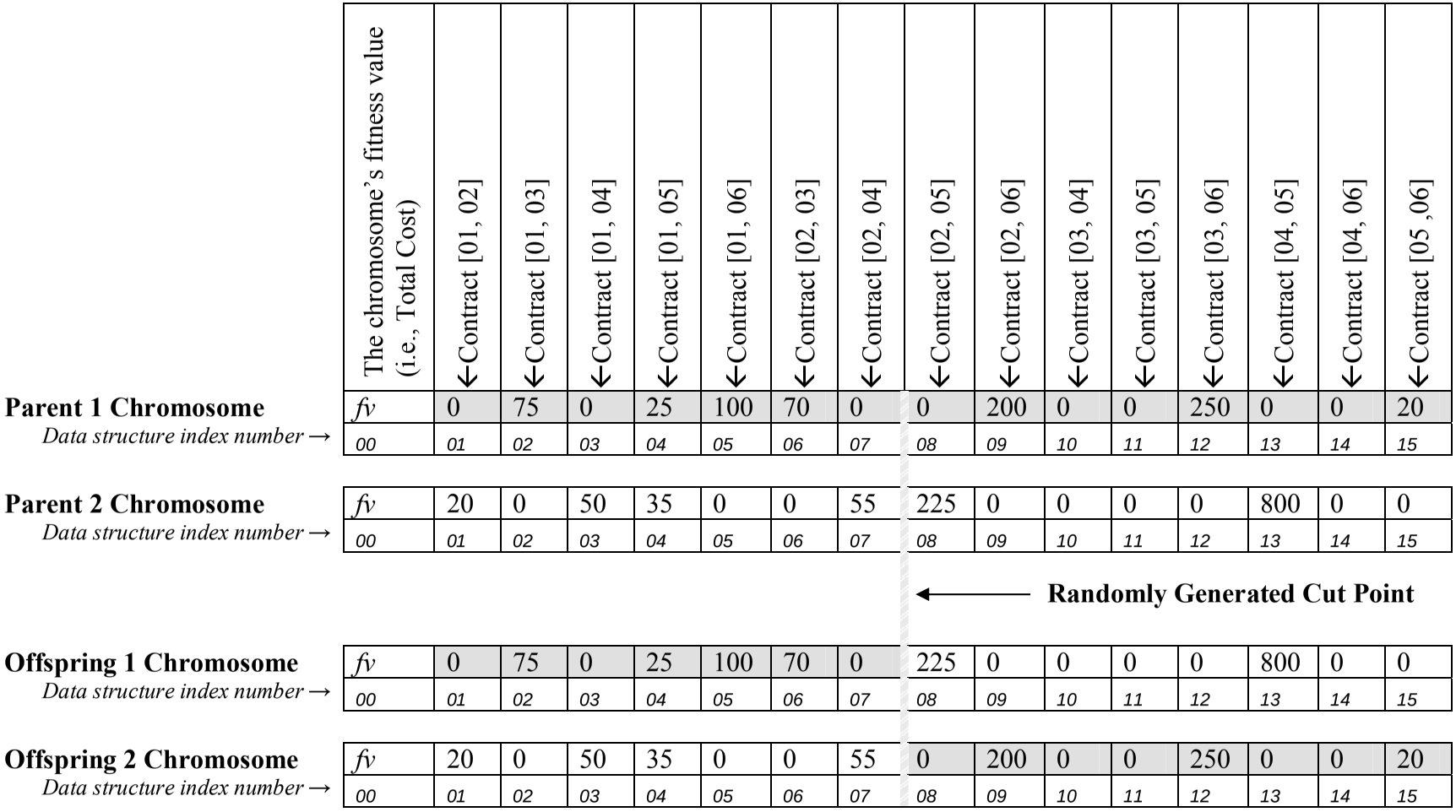


Figure 4-4: Example of the One-point Crossover Operator for $N = 5$

	The chromosome's fitness value (i.e., Total Cost)	←Contract [01, 02]	←Contract [01, 03]	←Contract [01, 04]	←Contract [01, 05]	←Contract [01, 06]	←Contract [02, 03]	←Contract [02, 04]	←Contract [02, 05]	←Contract [02, 06]	←Contract [03, 04]	←Contract [03, 05]	←Contract [03, 06]	←Contract [04, 05]	←Contract [04, 06]	←Contract [05, 06]
Parent 1 Chromosome	*fv*	0	75	0	25	100	70	0	0	200	0	0	250	0	0	20
Data structure index number →	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*
Parent 2 Chromosome	*fv*	20	0	50	35	0	0	55	225	0	0	0	0	800	0	0
Data structure index number →	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*
Randomly Binary Values: →	--	0	1	1	0	0	1	0	1	1	0	1	0	1	0	0
Data structure index number →	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*
Offspring 1 Chromosome	*fv*	20	75	0	35	0	70	55	0	200	0	0	0	0	0	0
Data structure index number →	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*
Offspring 2 Chromosome	*fv*	0	0	50	25	100	0	0	225	0	0	0	250	800	0	20
Data structure index number →	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*

Figure 4-5: Example of the Uniform Crossover Operator for $N = 5$

	The chromosome's fitness value (i.e., Total Cost)	←Contract [01, 02]	←Contract [01, 03]	←Contract [01, 04]	←Contract [01, 05]	←Contract [01, 06]	←Contract [02, 03]	←Contract [02, 04]	←Contract [02, 05]	←Contract [02, 06]	←Contract [03, 04]	←Contract [03, 05]	←Contract [03, 06]	←Contract [04, 05]	←Contract [04, 06]	←Contract [05, 06]
Parent Chromosome	*fv*	0	75	0	25	100	70	0	0	200	0	0	250	0	0	20
Data structure index number →	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*
Offspring Chromosome	*fv*	0	60	0	25	100	30	85	0	200	0	0	250	0	45	20
Data structure index number →	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*
Randomly Selected Genes →			↑				↑	↑							↑	

Figure 4-6: Example of the Mutation Operator for $N = 5$

4.1.4 Evaluation Function

The objective is to find a purchasing strategy that minimizes the total cost while satisfying demand for bandwidth as expressed in Equation (2). Equation (1) calculates the total cost for all purchased contracts over the planning horizon. The total cost is the chromosome's fitness value (*fv*). A chromosome's *fv* is recorded in Index 0 (see Figure 4-2). When evaluating a group of feasible chromosomes, chromosomes with lower *fv* are better than those with higher *fv*.

Recall from Equation (19) that $a_p = 13.116 + 962.927/Q + 258.526/L$. The numerical values in Equation (19) come from the regression analysis (shown in Table 2-5) for a BCP problem where *s* = Small. The general form of Equation (19) is $a_p = \beta_1 + \frac{\beta_2}{Q} + \frac{\beta_3}{L}$. For *s* = Medium and Large problems the numerical beta values in Equation (19) are different and are shown in Table 4-2. Notice that for Equations (24) and (26) the decision variable is $Q_{i,j}$. In these equations the $x_{i,j}$ binary variable is not needed since the *bws* ($Q_{i,j}$) of a contract indicates if it has been purchased.

The fitness value for the BCP-EA is the summation of the three cost components shown by Equations (24) through (26), expanded as

$$MIN\ TC = \sum_{i=1}^{N}\sum_{j=i+1}^{N+1} Q_{i,j}(j-i)\, r_p^{(i-1)}\beta_1 + \sum_{i=1}^{N}\sum_{j=i+1}^{N+1} x_{i,j}(j-i)\, r_p^{(i-1)}\beta_2 + \sum_{i=1}^{N}\sum_{j=i+1}^{N+1} Q_{i,j}\, r_p^{(i-1)}\beta_3 \quad , \tag{35}$$

which simplifies to

$$MIN\ TC = \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} Q_{i,j}\ r_p^{(i-1)}\ \left[(j-i)\beta_1 + \beta_3\right] + \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} x_{i,j}(j-i)\, r_p^{(i-1)} \beta_2\ . \qquad (36)$$

The feasibility of a solution is subject to the following constraints

$$\sum_{k=1}^{i} \sum_{j=i+1}^{N+1} Q_{k,j} \geq D_i \quad \rightarrow \quad \forall i = 1 \ldots N\ , \qquad (37)$$

which is the same as Equation (2), and

$$Q_{i,j} \leq W\ x_{i,j} \quad \forall i = 1 \ldots N;\ j = i+1 \ldots (N+1)\ , \qquad (38)$$

where

$$x_{i,j} \in \{0,1\}. \qquad (39)$$

4.1.5 Selection Method

The selection process used to obtain the next generation's population of chromosomes includes a generational policy, random selection, tournament selection, and an elitist strategy. The DBCP-EA uses an elitist strategy where a copy of the best chromosome identified within the *Pool_Array* at the end of a generation is copied in the next generation's population (i.e., the *Chromosome_Array*). If there is a tie, the best chromosome is selected randomly.

4.1.6 Generational Policy

A generational policy refers to how chromosomes are selected from one generation to the next. Throughout the run of any generation, some of the chromosomes undergo some alteration through the use of the crossover and the mutation operators. At the end of each generation, if the terminating condition is not met, a new population of

chromosomes is selected for the next generation. The new generation is selected from the *Pool_Array*.

Since the DBCP-EA uses an elitist strategy, it is possible that a 'good' chromosome could survive over many generations. The remaining 39 (*Pop_Size* – 1) chromosomes are selected using a tournament selection process. The tournament selection process begins by randomly selecting two chromosomes from the *Pool_Array* at the end of a generation. The *fv* of the two chromosomes are compared and the chromosome with the lowest *fv* wins. The winning chromosome is then added to the *Chromosome_Array* for the next generation.

Generation Policy Pseudo Code (i.e., procedure for selecting the beginning population of chromosomes for the next generation)

1. Check to see if the termination condition has been met. If true, there is no need to generate a population of chromosomes for the next generation since the program has ended. If false, go to step 2.
2. Check each chromosome in the *Pool_Array* for feasibility. If a chromosome is infeasible, the chromosome is made feasible by using the repair function (see Section 4.1.9). Calculate the *fv* for each chromosome in the *Pool_Array* and place the value in index 0 for each chromosome.
3. Perform an elitist strategy by copying the best chromosome in the *Pool_Array* into the *Chromosome_Array*. If there is a tie between the *fv* of two or more chromosomes in the *Pool_Array*, a best chromosome is randomly selected.
4. Perform a tournament selection (*Pop_Size* – 1) times on the chromosomes in the *Pool_Array*. *Count* = 0.
5. Randomly generate two numbers to select two chromosomes from the *Pool_Array*.
6. Perform a tournament selection by comparing the *fv* of the two chromosomes selected in step 5. The chromosome with the lowest *fv* is the winner. Add the winning chromosome into the *Chromosome_Array*. *Count* = *Count* + 1.
7. If *Count* < (*Pop_Size*-1), then go to step 5. Else, go to step 8.
8. Clear the *Pool_Array*, and then copy the chromosomes in the *Chromosome_Array* into the *Pool_Array*. The next generation's population has been selected. Done!

4.1.7 Termination Conditions

The DBCP-EA terminates as soon as one of two conditions is met. The first condition stipulates that if the best *fv* does not change for a predetermined number of generations, then the program will stop. This number is proportional to the number of periods (N) in the planning horizon. Specifically, if the DBCP EA runs Y_1 times, where $Y_1 = (N * 100)$, without finding a better solution, the program will terminate. Otherwise, processing will terminate when the program has completed a predetermined maximum number of generations $Y_2 = 5{,}000$. Y_1 and Y_2 were determined based on results from preliminary tests.

4.1.8 Computational Parameters

The size of each generation's population of chromosomes, *Pop_Size*, is held constant at 40 chromosomes. The probability of crossover and mutation were determined through preliminary tests. The *Uniform Crossover Rate*, O*ne-Point Crossover Rate*, and *Mutation Rate* are 0.8, 0.2, and 0.6, respectively.

4.1.9 Repair Function for Infeasible Solutions

The offspring produced by the three genetic operators are created by modifying and altering selected copies of chromosomes in a generation's beginning population. These offspring chromosomes could be infeasible, i.e., violate the constraint in Equation (37). If an offspring chromosome is infeasible it must be repaired prior to calculating its *fv*. Infeasible chromosomes are altered by the repair function to make them feasible.

The repair function has two separate repair algorithms used to change infeasible chromosomes into feasible chromosomes: RA_1 and RA_2. Infeasible solutions are repaired by both RA_1 and RA_2. Each algorithms employs a different rational in transforming

infeasible solutions into feasible solutions. A description of each algorithm is presented here, as well as the overall repair procedure.

The RA_1 algorithm calculates the bandwidth deficit for each period in the planning horizon starting backwards from period N. Let K equal the number of purchased contracts that affect the cumulative total bandwidth for a given period. Out of the K contracts, the most common contract (i.e., among the purchased contracts that affect a given period, the one that affects the greatest number of periods) has the period's deficit amount added to the *bws* for that contract. If there is a tie for the most common contract, a contract is randomly selected. Each period's deficit is then recalculated. If any period's deficit > 0, the latest period in the planning horizon that has a deficit is identified. Again, the period's deficit amount is added to the period's most common contract. This procedure of deficit elimination continues until there are no more deficits.

The rational for RA_1 stems from the observation that when s = Large, p = Small or Medium, and d = Small or Medium the lowest *fv* often occurs when the last contract in each period is purchased. Therefore, RA_1 takes advantage of contract size discounts of active contracts that will not only satisfy the *bws* requirements for a given period, but will also increase the *bws* for the most additional periods.

Within the chromosome data structure, for each period the shortest contract is listed first, and the longest contract is listed last. The longer the duration of a contract, the more subsequent periods that will include that *bws*. Therefore, longer duration contracts, which affect a greater number of periods, will be identified as most common contracts. Therefore, adding deficit amounts to a most common contract affects the available bandwidth for not only the period in which the deficit has been identified, but

also all subsequent period for which the contract is active. For example, Figure 4-7 shows an $N = 5$ period chromosome. The first row of white cells for each period represents the period in which that contract can be purchased. The other grey cells represent the subsequent period for which a contract *bws* is active.

A purchased contract is active for a number of periods equal to its duration. As illustrated in Figure 4-7, Contract [01, 03], having a *bws* of 500 Mbps, can only be purchased in period 1. This contract has a length of 2 periods (i.e., ending at the beginning of period 3 – starting at the beginning of period 1) therefore this contract provides 500 Mbps in period 1, as well as 500 Mbps in period 2.

Figure 4-7 shows a deficit of 100 Mbps in period 5. The most common active contract would be Contract [01, 06]. Starting backwards with Period 5, the active contracts are [01, 06], [04, 06], and [05, 06]. The duration for each of these contracts is calculated and the contract with the longest duration is selected to receive the deficit amount. For example, Contract [01, 06] is active for all 5 periods; therefore its duration is 5. Contract [04, 06] has duration of 2, and Contract [05, 06] has duration of 1. Of these contract lengths, duration of 5 is the largest; therefore Contract [01, 06] is the most common contract.

For the example shown in Figure 4-7, algorithm RA_1 adds the deficit amount of 100 Mbps to the *bws* already recorded in Contract [01, 06]. Notice that since Contract [01, 06] is purchased in period 1 and it is active for all 5 period its *bws* is added to the available bandwidth for periods 1 through 5. Therefore, the infeasible solution shown in Figure 4-7 would be repaired as shown in Figure 4-8.

Period	Contract Length	Contract Number														
		Planning Horizon (N = 5 periods)														
		1	*2*	*3*	*4*	*5*	*1*	*2*	*3*	*4*	*1*	*2*	*3*	*1*	*2*	*1*
1		0	0	0	0	500										
2			0	0	0	500	300	0	0	0						
3				0	0	500		0	0	0	0	600	0			
4					0	500			0	0		600	0	0	800	
5						500				0			0		800	900
		Contract [01, 02]	Contract [01, 03]	Contract [01, 04]	Contract [01, 05]	Contract [01, 06]	Contract [02, 03]	Contract [02, 04]	Contract [02, 05]	Contract [02, 06]	Contract [03, 04]	Contract [03, 05]	Contract [03, 06]	Contract [04, 05]	Contract [04, 06]	Contract [05, 06]

(a) Cumulative Contracted Bandwidth per Period	(b) Required Bandwidth per Period	(c) Period Deficit (i.e., (c) = (a) – (b) iff ((a) – (b)) > 0)
500	500	0
800	880	80
1,100	1,100	0
1,900	1,900	0
2,200	2,300	100

Figure 4-7: Example of the RA_1 Repair Algorithm

Period	Contract Length	Contract Number	Contract [01, 02]	Contract [01, 03]	Contract [01, 04]	Contract [01, 05]	Contract [01, 06]	Contract [02, 03]	Contract [02, 04]	Contract [02, 05]	Contract [02, 06]	Contract [03, 04]	Contract [03, 05]	Contract [03, 06]	Contract [04, 05]	Contract [04, 06]	Contract [05, 06]
			Planning Horizon (N = 5 periods)														
1	*1*		0	0	0	0	**600**										
2	*2*			0		0	**600**	300	0	0	0						
3	*3*				0	0	**600**		0	0	0	0	600	0			
4	*4*					0	**600**			0	0		600	0	0	800	
5	*5*						**600**				0			0	0	800	900

(a) Cumulative Contracted Bandwidth per Period	(b) Required Bandwidth per Period	(c) Period Deficit (i.e., (c) = (a) – (b) iff ((a) – (b)) > 0)
600	500	0
900	880	**0**
1,100	1,100	0
1,900	1,900	0
2,300	2,300	**0**

Figure 4-8: *RA1* Example after Increasing Contract [01, 06] Bandwidth Size by 100 Mbps

In Figures 4-7 and 4-8 the white cells in each row indicate those contracts that could be active during each period. The total bandwidth for any period is equal to the total bandwidth purchased at the beginning of each period, plus the cumulative bandwidth of any pervious contracts that are active during each period. Therefore, to calculate the available bandwidth for a period you simply add the values in the white cells for the newly purchased bandwidth, plus the values in the grey cells for each period's row. These are the values recorded in the columns labeled "(a) Cumulative Contracted Bandwidth per Period."

The RA_2 algorithm uses a backward period-by-period random contract deficit elimination procedure. RA_2 alters selected contracts by moving backwards from period N to period 1. The rational for eliminating deficits by moving backwards is that price decreases and demand increases over the planning horizon. By starting with the later periods and working backwards the lowest period prices are used first, just as in RA_1. RA_2 calculates the bandwidth deficit for each period. Starting backwards, this repair algorithm randomly adds a period's deficit to a randomly selected contract that is active during that period. After randomly selecting a contract and adding the deficit, the *fv* is recalculated. This process is repeated 20 times to encourage exploration of the search space, while at the same time limiting search time. Out of the 20 repaired chromosomes, the repaired chromosome with the lowest *fv* is selected to be the final repaired chromosome for the RA_2 repair algorithm.

The two repaired and feasible chromosomes resulting from RA_1 and RA_2 are then compared and the one with the lowest *fv* replaces the original infeasible chromosome in the *Pool_Array*. Note that RA_1 takes advantage of knowledge of the BCP, while RA_2

benefits from randomness. Both algorithms have their advantages; however, they are both biased because they both begin the alteration process from the latest period in the planning horizon and work backward. Future research may identify better, less biased repair algorithms. The pseudo code listed below outlines in detail the repair function, including the two repair algorithms, RA_1 and RA_2.

Repair Function Pseudo Code

(0.1) After the three genetic operators (see Section 4.1.3) have been performed. *Count* = (0). Set *Pool_Size* = (Pop_Size + the number of Offspring created in last generation).

(0.2) *Count* = *Count* + 1.

(0.3) Select the chromosome number *Count* from the *Pool_Array*, and check for feasibility. If the selected chromosome is infeasible, go to step (0.4). Else if *Count* = *Pool_Size*, go to step (0.6). Else if *Count* < *Pool_Size*, go to step (0.2).

(0.4) Copy the infeasible chromosome into the temporary holding arrays: *Holder1* used in RA_1), *Holder2* (used in RA_2), and *Best_Holder* (also used in RA_2).

- Begin RA_1: Use *Holder1* to complete steps (1.1) through (1.8).

  (1.1) *Count2* = *N* + 1.

  (1.2) *Count2* = *Count2* – 1. .

  (1.3) Determine if there is a deficit for period *Count2*. If yes, then go to step (1.4). Else, go to step (1.2).

  (1.4) Record all of the purchased contracts that affect the cumulative available bandwidth total for period *Count2*.

  (1.5) For each contract recorded in step (1.4), count how many periods the contract is active.

  (1.6) For period *Count2*, find the most common contract (i.e., an active contract that not only affects period *Count2*, but also affects the most number of other periods. Ties are broken randomly.). Once a contract is selected, determine the period in which that contract can be purchased.

  (1.7) Add the deficit amount for period *Count2* to the *bws* for the contract selected in step (1.6). If *Count2* = 1, go to step (1.8). Else, go to step (1.2).

  (1.8) Calculate the *fv* for the chromosome in *Holder1* and enter that value in index 0. RA_1 finished; go to step (2.1).

Begin RA_2: Used *Holder2* and *Best_Holder* to compete steps (2.1) through (2.8). *Count1* = 0.

(2.1) *Count1* = *Count1* + 1. *Count2* = *N* + 1.

(2.2) While *Count1* ≤ 20 go to step (2.3). If *Count1* > 20, go to step (2.8).

(2.3) *Count2* = *Count2* – 1.

(2.4) For period *Count2*, determine if there is a bandwidth deficit. If period *Count2* has a deficit, then record all the contracts that could affect period *Count2*.

(2.5) Let *M* = the number of active contracts in period *Count2*. Randomly generate a number between 1 and *M* to select a contract, and add the deficit amount for period *Count2* to the selected contract's *bws*.

(2.6) If *Count2* > 1, go to step (2.3). Else, go to step (2.7).

(2.7) Calculate the *fv* for the chromosome in *Holder2* and enter that value in index 0. If *Count1* = 1, then copy the chromosome in *Holder2* into *Best_Holder*. If *Count1* > 1, then compare the *fv* in index 0 of *Best_Holder* to the *fv* in *Holder2*. If the *fv* in *Holder2* < *Best_Holder*, then copy the chromosome in *Holder2* into *Best_Holder*.

(2.8) If *Count1* < 21, copy the original chromosome at location *Count* (see step (0.2)) from the *Pool_Array* into *Holder2* and go to step (2.1). If *Count1* = 20, then RA_2 is finished; go to step (0.5).

(0.5) Compare the *fv* for *Holder1* and *Holder2*. The chromosome with the lowest *fv* replaces the original chromosome in the *Pool_Array* at location *Count* (see step (0.2)). If *Count* < *Pool_Size*, go to step (0.3). Else, go to step (0.6).

(0.6) Done! All infeasible chromosomes in the *Pool_Array* have been repaired and each has had its *fv* calculated and recorded in index 0.

4.2 DBCP-EA Experimental Design

To evaluate the performance of DBCP-EA, we consider the impact of four parameters: planning horizon size (*N*), temporal discount (*s*), price decrease rate (*p*), and demand increase rate (*d*) (see Figure 4-9). Each parameter has three values, Small, Medium and Large (see Table 4-1), resulting in 81 problems.

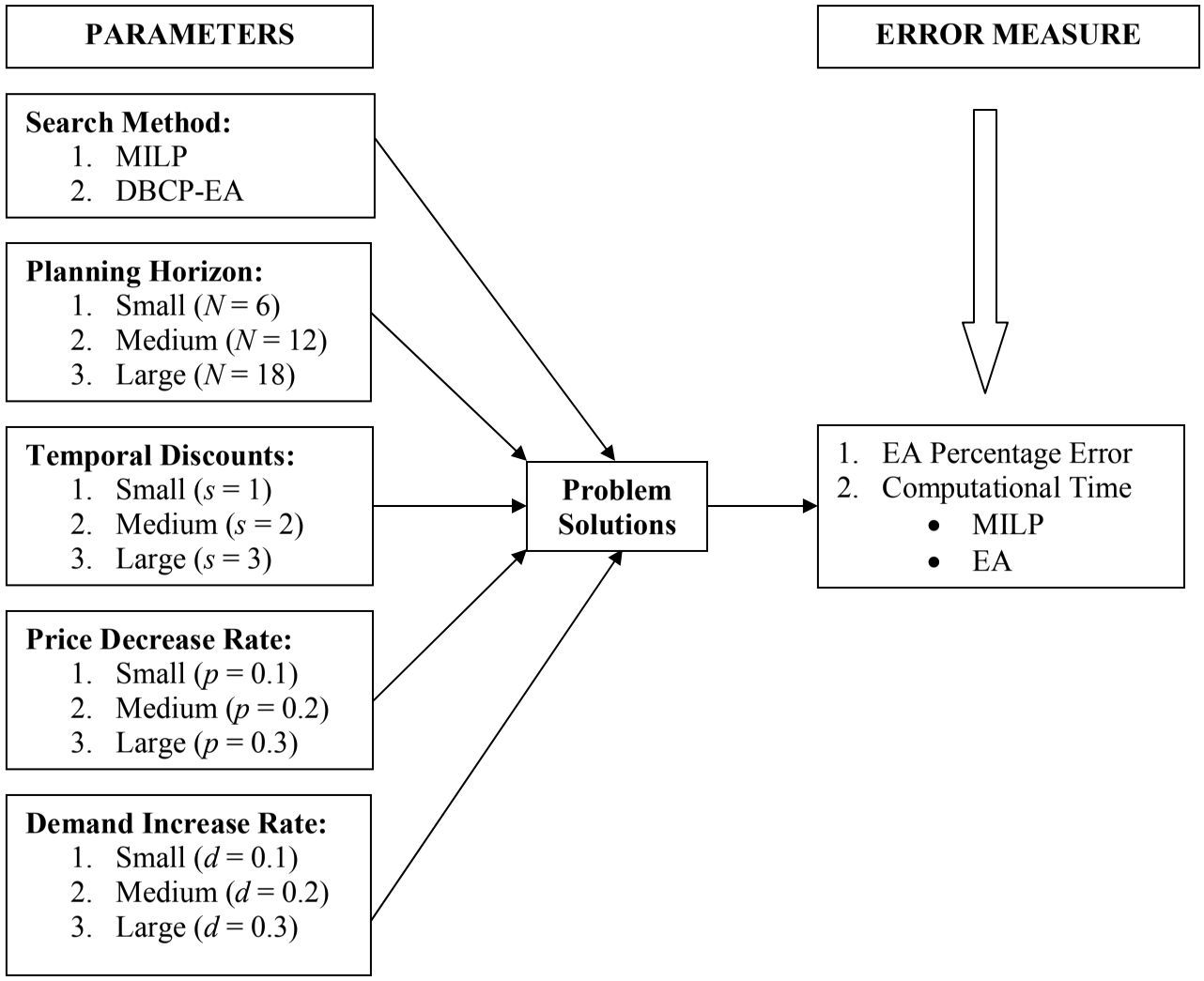


Figure 4-9: Experimental Design for the DBCP-EA

Two solution methods are employed: (1) MILP, and (2) the DBCP-EA. MILP gives the global optimal solution and can be used as a benchmark to measure the quality of the DBCP-EA solutions. The *%_Error* is the percent increase in the total purchasing cost of the best solution found by the DBCP-EA over the optimal solution identified by MILP. The computational time in seconds to solve a problem using the DBCP-EA and MILP is a secondary measure of comparison.

Table 4-1: The Three Values for each of the Four Parameters: *N*, *s*, *d*, and *p*

Parameters	Small	Medium	Large
N	6	12	18
s	1	2	3
d	0.1	0.2	0.3
p	0.1	0.2	0.3

Table 4-1 shows that the three levels of temporal discount (*s*) represented in the computer code as 1, 2, and 3 are referred to here as Small, Medium, and Large, respectively. Temporal discounts represent the discounts per Mbps for longer duration contracts. Table 4-2 summarizes the data related to the three levels of *s*. The bandwidth cost data for *s* = Small was presented in Table 1-2. For a 2 year contract, the bandwidth price for the same bandwidth size is 95% of the cost for a 1 year contract; 3 year contracts are 85% of the cost of a 1 year contract. For problems where *s* = Medium, Table 4-2 shows the cost per Mbps of bandwidth for 2 years is 90% of that for a 1 year contract; the bandwidth cost for a 3 year contract is 80% of that for a 1 year contract. When *s* = Large the discount rates are 85% and 75% for 2 and 3 year contracts, respectively. As can be seen from Table 4-2 the regression performed well for all levels of *s*. The resulting regression beta coefficients are used in regression equations to calculate the total cost for a purchasing strategy given the bandwidth size (*Q*) and

duration (L) for each active contract in a planning horizon for each of the three temporal discount levels.

Table 4-2: Summary of the Temporal Discount Data

Temporal Discount Level (s)	Contract Duration (years)			Adjusted R Squared	Alpha Sign. Level	Regression Equation Coefficients†		
		Duration Discounts						
	1*	2	3			β_1	β_2	β_3
Small	c	c*95%	c*85%	**98.8%**	**1.0 E-17**	13.12	962.93	258.53
Medium	c	c*90%	c*80%	98.6%	1.0 E-17	6.76	928.50	364.99
Large	c	c*85%	c*75%	98.2%	1.0 E-16	0.40	894.11	471.42

*c = The cost data in the column labeled "1 year" from **Table 1-2**
†**Equation (19)** is the regression equation

4.3 DBCP-EA Example Problems

We use three example problems, one for each N, where s = Small, p = Large, and d = Small, to compare the solutions derived by the DBCP-EA and the MILP. Figure 4-10 shows the general MILP formulation that is used to generate the MILP input file. Table 4-3 shows the MILP input data and Table 4-4 shows the MILP solution for the smallest example problem, (N = Small, s = Small, p = Large, and d = Small). Contracts with a binary variable $x_{i,j}$ = 1 are the purchased contracts for the purchasing strategy described in a solution.

$$\text{MIN} \sum_{i=1}^{N} \sum_{j=i+1}^{N+1} \left((\text{TC}_1 + TC_3) Q_{i,j} + TC_2 \; x_{i,j} \right)$$

ST

$$\sum_{i=1}^{N} \sum_{j=i+1}^{N+1} Q_{i,j} \geq D_i$$

$$Q_{i,j} - W \; x_{i,j} < 0 \quad \forall i = 1 \ldots N, \forall j = 2 \ldots (N+1), \text{where } W \text{ is a large number}$$

END

$$\text{INT } x_{i,j} \quad \forall i = 1 \ldots N, \forall j = 2 \ldots (N+1)$$

Figure 4-10: MILP Formulation

Table 4-3: The *N* = Small Example Problem MILP Input Data

Example Problem: *N* = Small, *s* = Small, *p* = Large, *d* = Small	
Objective Function	Z = MIN 271.64 Q[01, 02] + 962.93 X[01, 02] + 284.76 Q[01, 03] + 1925.85 X[01, 03] + 297.87 Q[01, 04] + 2888.78 X[01, 04] + 310.99 Q[01, 05] + 3851.71 X[01, 05] + 324.11 Q[01, 06] + 4814.64 X[01, 06] + 337.22 Q[01, 07] + 5777.56 X[01, 07] + 190.15 Q[02, 03] + 674.05 X[02, 03] + 199.33 Q[02, 04] + 1348.10 X[02, 04] + 208.51 Q[02, 05] + 2022.15 X[02, 05] + 217.69 Q[02, 06] + 2696.20 X[02, 06] + 226.87 Q[02, 07] + 3370.24 X[02,07] + 133.10 Q[03, 04] + 471.83 X[03, 04] + 139.53 Q[03, 05] + 943.67 X[03, 05] + 145.96 Q[03, 06] + 1415.50 X[03, 06] + 152.39 Q[03, 07] + 1887.34 X[03, 07] + 93.17 Q[04, 05] + 330.28 X[04, 05] + 97.67 Q[04, 06] + 660.57 X[04, 06] + 102.17 Q[04, 07] + 990.85 X[04, 07] + 65.22 Q[05, 06] + 231.20 X[05, 06] + 68.37 Q[05, 07] + 462.40 X[05, 07] + 45.65 Q[06, 07] + 161.84 X[06, 07]
Subject to the following constraints:	
Demand Requirement Constraint	Q[01, 02] + Q[01, 03] + Q[01, 04] + Q[01, 05] + Q[01, 06] + Q[01, 07] > 540.00
	Q[01, 03] + Q[01, 04] + Q[01, 05] + Q[01, 06] + Q[01, 07] + Q[02, 03] + Q[02, 04] + Q[02, 05] + Q[02, 06] + Q[02, 07] > 594.00
	Q[01, 04] + Q[01, 05] + Q[01, 06] + Q[01, 07] + Q[02, 04] + Q[02, 05] + Q[02, 06] + Q[02, 07] + Q[03, 04] + Q[03, 05] + Q[03, 06] + Q[03, 07] > 654.00
	Q[01, 05] + Q[01, 06] + Q[01, 07] + Q[02, 05] + Q[02, 06] + Q[02, 07] + Q[03, 05] + Q[03, 06] + Q[03, 07] + Q[04, 05] + Q[04, 06] + Q[04, 07] > 719.00
	Q[01, 06] + Q[01, 07] + Q[02, 06] + Q[02, 07] + Q[03, 06] + Q[03, 07] + Q[04, 06] + Q[04, 07] + Q[05, 06] + Q[05, 07] > 791.00
	Q[01, 07] + Q[02, 07] + Q[03, 07] + Q[04, 07] + Q[05, 07] + Q[06, 07] > 870.00

Non-negative Contract Size Constraint		Binary Variable associated with each Contract	
Q[01, 02] - 1080.00 X[01, 02] < 0	Q[03, 04] - 1308.00 X[03, 04] < 0	INT X[01, 02]	INT X[03, 04]
Q[01, 03] - 1188.00 X[01, 03] < 0	Q[03, 05] - 1438.00 X[03, 05] < 0	INT X[01, 03]	INT X[03, 05]
Q[01, 04] - 1308.00 X[01, 04] < 0	Q[03, 06] - 1582.00 X[03, 06] < 0	INT X[01, 04]	INT X[03, 06]
Q[01, 05] - 1438.00 X[01, 05] < 0	Q[03, 07] - 1740.00 X[03, 07] < 0	INT X[01, 05]	INT X[03, 07]
Q[01, 06] - 1582.00 X[01, 06] < 0	Q[04, 05] - 1438.00 X[04, 05] < 0	INT X[01, 06]	INT X[04, 05]
Q[01, 07] - 1740.00 X[01, 07] < 0	Q[04, 06] - 1582.00 X[04, 06] < 0	INT X[01, 07]	INT X[04, 06]
Q[02, 03] - 1188.00 X[02, 03] < 0	Q[04, 07] - 1740.00 X[04, 07] < 0	INT X[02, 03]	INT X[04, 07]
Q[02, 04] - 1308.00 X[02, 04] < 0	Q[05, 06] - 1582.00 X[05, 06] < 0	INT X[02, 04]	INT X[05, 06]
Q[02, 05] - 1438.00 X[02, 05] < 0	Q[05, 07] - 1740.00 X[05, 07] < 0	INT X[02, 05]	INT X[05, 07]
Q[02, 06] - 1582.00 X[02, 06] < 0	Q[06, 07] - 1740.00 X[06, 07] < 0	INT X[02, 06]	INT X[06, 07]
Q[02, 07] - 1740.00 X[02, 07] < 0		INT X[02, 07]	

Table 4-4: The *N* = Small Example Problem MILP Solution

Example Problem: *N* = Small, *s* = Small, *p* = Large, *d* = Small						
Objective value: $231,313.5						
Period Number	**Contract Duration**	**Contract Number**	**Bandwidth Size**		**Contract Number Binary Variable**	**Binary Value (i.e., 1 = Purchased Contract)**
1	1	Q[01, 02]	0		X[01, 02]	0
1	2	Q[01, 03]	0		X[01, 03]	0
1	3	Q[01, 04]	0		X[01, 04]	0
1	4	Q[01, 05]	0		X[01, 05]	0
1	5	Q[01, 06]	0		X[01, 06]	0
1	6	Q[01, 07]	540	⇨	X[01, 07]	1
2	1	Q[02, 03]	0		X[02, 03]	0
2	2	Q[02, 04]	0		X[02, 04]	0
2	3	Q[02, 05]	0		X[02, 05]	0
2	4	Q[02, 06]	0		X[02, 06]	0
2	5	Q[02, 07]	54	⇨	X[02, 07]	1
3	1	Q[03, 04]	0		X[03, 04]	0
3	2	Q[03, 05]	0		X[03, 05]	0
3	3	Q[03, 06]	0		X[03, 06]	0
3	4	Q[03, 07]	60	⇨	X[03, 07]	1
4	1	Q[04, 05]	0		X[04, 05]	0
4	2	Q[04, 06]	0		X[04, 06]	0
4	3	Q[04, 07]	65	⇨	X[04, 07]	1
5	1	Q[05, 06]	0		X[05, 06]	0
5	2	Q[05, 07]	72	⇨	X[05, 07]	1
6	1	Q[06, 07]	79	⇨	X[06, 07]	1

Figure 4-11 shows all the contracts that could be active during each period. Contract [01, 07] is purchased in period 1 with a *bws* of 540 Mbps and has a length of 6 periods; therefore 540 Mbps is available for all 6 periods. The bandwidth available in period 2 is 594 Mbps which is the sum of bandwidths purchased for all contracts active during period 2 (Contract [02, 07] and Contract [01, 07]).

Period	Contract Number	Contract [01, 02]	Contract [01, 03]	Contract [01, 04]	Contract [01, 05]	Contract [01, 06]	Contract [01, 07]	Contract [02, 03]	Contract [02, 04]	Contract [02, 05]	Contract [02, 06]	Contract [02, 07]	Contract [03, 04]	Contract [03, 05]	Contract [03, 06]	Contract [03, 07]	Contract [04, 05]	Contract [04, 06]	Contract [04, 07]	Contract [05, 06]	Contract [05, 07]	Contract [06, 07]	Available Bandwidth per Period	Required Demand per Period	Bandwidth Deficit
1		0	0	0	0	0	**540**																540	540	0
2			0	0	0	0	**540**	0	0	0	0	**54**											594	594	0
3				0	0	0	**540**		0	0	0	**54**	0	0	0	**60**							654	654	0
4					0	0	**540**			0	0	**54**		0	0	**60**	0	0	**65**				719	719	0
5						0	**540**				0	**54**			0	**60**		0	**65**	0	**72**		791	791	0
6							**540**					**54**				**60**			**65**		**72**	**79**	870	870	0

Figure 4-11: Diagram of MILP Example Problem Solution for N = Small, s = Small, p = Large, d = Small

Table 4-5 compares the solutions generated by the MILP and the DBCP-EA for each of the three example problems. The *%_Error* value is used to measure out how close the DBCP-EA solution is from the optimal solution. The *%_Error* value is calculated by subtracting the DBCP-EA solution's total cost from the MILP solution's total cost and then divided by the MILP solution's total cost and multiplied by 100.The DBCP-EA was able to find optimal solutions (*%_Errors* = 0) for *N* = Small and *N* = Medium problems. However, for *N* = Large the DBCP-EA solution had a *%_Error* = 0.25%.

Table 4-5: Cost and Error, and Time Data for Example Problems

Problem: *N* = Small, *s* = Small, *p* = Large, *d* = Small		
Cumulative Costs for Each Solution	MILP Solution Total Cost =	$231,313.45
	Best EA Solution Total Cost =	$231,314.00
Error Data for the EA Solution	Cost Difference* =	$0.50
	%_Error =	0.00
Computational Time for Each Solution	MILP Time (seconds) =	0.010
	EA Time (seconds) for Best EA =	0.072
Problem: *N* = Medium, *s* = Small, *p* = Large, *d* = Small		
Cumulative Costs for Each Solution	MILP Solution Total Cost =	$276,257.67
	Best EA Solution Total Cost =	$276,261.00
Error Data for the EA Solution	Cost Difference* =	$3.30
	%_Error =	0.00
Computational Time for Each Solution	MILP Time (seconds) =	7.590
	EA Time (seconds) for Best EA =	73.547
Problem: *N* = Large, *s* = Small, *p* = Large, *d* = Small		
Cumulative Costs for Each Solution	MILP Solution Total Cost =	$285,071.26
	Best EA Solution Total Cost =	$285,777.00
Error Data for the EA Solution	Cost Difference =	$705.74
	%_Error =	0.25
Computational Time for Each Solution	MILP Time (seconds) =	1,373.000
	EA Time (seconds) for Best EA =	1,338.562

* Cost Difference due to precision and rounding errors

Table 4-5 also shows that as *N* increases the computational time between the MILP and the DBCP-EA becomes comparable, especially for the *N* = Large problem

where the DBCP-EA found a solution in less time than the MILP. It was originally planned to use 6 month increments for *N* ranging from 6 to 24, but CPLEX was unable to find a solution to the MILP for a planning horizon of 24 months within a reasonable amount of time. Furthermore, it should be noted that the DBCP-EA was not designed with the objective of minimizing computational time. Therefore it may be possible in the future to re-code the DBCP-EA to improve computational.

Tables 4-6 through 4-8 show a period by period breakdown of the MILP and the DBCP-EA solutions for the three example problems. The column labeled "Available Bandwidth" shows the total bandwidth available in for any given period. Tables 4.6 and 4.7 show the DBCP-EA solutions for *N* = Small and *N* = Medium. In these two example problems there is no surplus bandwidth in any period for these two problems, and the constraint from Equation (2) is binding. Moreover, by comparing the "Contract Number" columns for the MILP and the DBCP-EA solutions theses tables show that both search methods selected the same contracts for each period. Since the MILP gives an optimal solution, we can assert that the DBCP-EA has identified an optimal, lowest total cost solution for the *N* = Small and Medium example problems.

Table 4-6: MILP vs. DBCP-EA Purchasing Strategy Solutions for N = Small, s = Small, p = Large, and d = Small

Constraint		**MILP Solution**			**EA Solution**			**Same Contract Number**	**Same Available *bws***
Period Number	**Required Demand per Period from Equation (2)**	**(a) Contract Number**	**Bandwidth Size (*bws*)**	**(b) Available Bandwidth per Period**	**(c) Contract Number**	**Bandwidth Size (*bws*)**	**(d) Available Bandwidth per Period**	**(a) = (c)**	**(b) = (d)**
1	540	[01, 07]	540	540	[01, 07]	540	540	●	●
2	594	[02, 07]	54	594	[02, 07]	54	594	●	●
3	654	[03, 07]	60	654	[03, 07]	60	654	●	●
4	719	[04, 07]	65	719	[04, 07]	65	719	●	●
5	791	[05, 07]	72	791	[05, 07]	72	791	●	●
6	870	[06, 07]	79	870	[06, 07]	79	870	●	●

Table 4-7: MILP vs. DBCP-EA Purchasing Strategy Solutions for N = Medium, s = Small, p = Large, and d = Small

Constraint		**MILP Solution**			**EA Solution**			**Same Contract Number**	**Same Available *bws***
Period Number	**Required Demand per Period from Equation (2)**	**(a) Contract Number**	**Bandwidth Size (*bws*)**	**(b) Available Bandwidth per Period**	**(c) Contract Number**	**Bandwidth Size (*bws*)**	**(d) Available Bandwidth per Period**	**(a) = (c)**	**(b) = (d)**
1	540	[01, 07]	540	540	[01, 07]	540	540	●	●
2	594	[02, 06]	54	594	[02, 06]	54	594	●	●
3	654	[03, 07]	60	654	[03, 07]	60	654	●	●
4	719	[04, 08]	65	719	[04, 08]	65	719	●	●
5	791	[05, 09]	72	791	[05, 09]	72	791	●	●
6	870	[06, 13]	133	870	[06, 13]	133	870	●	●
7	957	[07, 13]	687	957	[07, 13]	687	957	●	●
8	1,053	[08, 13]	161	1,053	[08, 13]	161	1,053	●	●
9	1,158	[09, 13]	177	1,158	[09, 13]	177	1,158	●	●
10	1,274	[10, 13]	116	1,274	[10, 13]	116	1,274	●	●
11	1,401	[11, 13]	127	1,401	[11, 13]	127	1,401	●	●
12	1,541	[12, 13]	140	1,541	[12, 13]	140	1,541	●	●

Table 4.8 shows the *N* = Large example problem. The MILP and the DBCP-EA did not find the same solution, although in each solution constraint Equation (2) is binding, therefore both solutions are feasible. In the MILP solution the available bandwidth is exactly equal to the required demand for each period, and is the same for both the MILP and DBCP-EA solution; however, the contracts purchased do not match. The periods where purchased contracts are the same can be seen in the column labeled "Same Contract Number." The *%_Error* between the MILP and the DBCP-EA solutions is 0.25%. Therefore, the DBCP-EA solution represents a feasible solution, albeit not optimal. This solution could then be "tweaked" if desired. It should also be noted that if the DBCP-EA had been allowed to run for a longer time it is possible that the optimal solution might have been found.

Table 4-8: MILP vs. DBCP-EA Purchasing Strategy Solutions for N = Large, s = Small, p = Large, and d = Small

Constraint		**MILP Solution**			**EA Solution**			**Same Contract Number**	**Same Available *bws***
Period Number	**Required Demand per Period from Equation (2)**	**(a) Contract Number**	**Bandwidth Size (*bws*)**	**(b) Available Bandwidth per Period**	**(c) Contract Number**	**Bandwidth Size (*bws*)**	**(d) Available Bandwidth per Period**	**(a) = (c)**	**(b) = (d)**
1	540	[01, 07]	540	540	[01, 07]	540	540	●	●
2	594	[02, 06]	54	594	[02, **07**]	54	594		●
3	654	[03, 07]	60	654	[03, **08**]	60	654		●
4	719	[04, 08]	65	719	[04, 08]	65	719	●	●
5	791	[05, 10]	72	791	[05, **11**]	72	791		●
6	870	[06, 12]	133	870	[06, 12]	79	870	●	●
7	957	[07, 13]	687	957	[07, **14**]	681	957		●
8	1,053	[08, 13]	161	1,053	[08, **12**]	221	1,053		●
9	1,158	[09, 14]	105	1,158	[09, **13**]	105	1,158		●
10	1,274	[10, 15]	188	1,274	[10, **16**]	116	1,274		●
11	1,401	[11, 16]	127	1,401	[11, **19**]	199	1,401		●
12	1,541	[12, 19]	273	1,541	[12, 19]	440	1,541	●	●
13	1,695	[13, 19]	1,002	1,695	[13, 19]	259	1,695	●	●
14	1,865	[14, 19]	275	1,865	[14, 19]	851	1,865	●	●
15	2,051	[15, 19]	374	2,051	[15, 19]	186	2,051	●	●
16	2,256	[16, 19]	332	2,256	[16, 19]	321	2,256	●	●
17	2,482	[17, 19]	226	2,482	[17, 19]	226	2,482	●	●
18	2,730	[18, 19]	248	2,730	[18, 19]	248	2,730	●	●

4.4 DBCP-EA Results

Table 4-9 shows the parameter settings for each of the 81 problems identified in the experimental design in Section 4.2. The categories in the table stand for Small (Sm), Medium (Md) and Large (Lg). There are a total of 81 problems (4 parameters with 3 categories each).

Table 4-9: Problem Number Corresponding to Each of the 81 Problems

#	*N*	*S*	*p*	*d*	#	*N*	*s*	*p*	*d*	#	*N*	*s*	*p*	*d*
1	Sm	Sm	Sm	Sm	28	Md	Sm	Sm	Sm	55	Lg	Sm	Sm	Sm
2	Sm	Sm	Sm	Md	29	Md	Sm	Sm	Md	56	Lg	Sm	Sm	Md
3	Sm	Sm	Sm	Lg	30	Md	Sm	Sm	Lg	57	Lg	Sm	Sm	Lg
4	Sm	Sm	Md	Sm	31	Md	Sm	Md	Sm	58	Lg	Sm	Md	Sm
5	Sm	Sm	Md	Md	32	Md	Sm	Md	Md	**59**	**Lg**	**Sm**	**Md**	**Md**
6	Sm	Sm	Md	Lg	33	Md	Sm	Md	Lg	**60**	**Lg**	**Sm**	**Md**	**Lg**
7	Sm	Sm	Lg	Sm	34	Md	Sm	Lg	Sm	61	Lg	Sm	Lg	Sm
8	Sm	Sm	Lg	Md	35	Md	Sm	Lg	Md	62	Lg	Sm	Lg	Md
9	Sm	Sm	Lg	Lg	**36**	**Md**	**Sm**	**Lg**	**Lg**	**63**	**Lg**	**Sm**	**Lg**	**Lg**
10	Sm	Md	Sm	Sm	37	Md	Md	Sm	Sm	64	Lg	Md	Sm	Sm
11	Sm	Md	Sm	Md	38	Md	Md	Sm	Md	65	Lg	Md	Sm	Md
12	Sm	Md	Sm	Lg	39	Md	Md	Sm	Lg	66	Lg	Md	Sm	Lg
13	Sm	Md	Md	Sm	40	Md	Md	Md	Sm	**67**	**Lg**	**Md**	**Md**	**Sm**
14	Sm	Md	Md	Md	41	Md	Md	Md	Md	**68**	**Lg**	**Md**	**Md**	**Md**
15	Sm	Md	Md	Lg	42	Md	Md	Md	Lg	**69**	**Lg**	**Md**	**Md**	**Lg**
16	Sm	Md	Lg	Sm	43	Md	Md	Lg	Sm	70	Lg	Md	Lg	Sm
17	Sm	Md	Lg	Md	**44**	**Md**	**Md**	**Lg**	**Md**	**71**	**Lg**	**Md**	**Lg**	**Md**
18	Sm	Md	Lg	Lg	**45**	**Md**	**Md**	**Lg**	**Lg**	**72**	**Lg**	**Md**	**Lg**	**Lg**
19	Sm	Lg	Sm	Sm	46	Md	Lg	Sm	Sm	73	Lg	Lg	Sm	Sm
20	Sm	Lg	Sm	Md	47	Md	Lg	Sm	Md	74	Lg	Lg	Sm	Md
21	Sm	Lg	Sm	Lg	48	Md	Lg	Sm	Lg	75	Lg	Lg	Sm	Lg
22	Sm	Lg	Md	Sm	49	Md	Lg	Md	Sm	76	Lg	Lg	Md	Sm
23	Sm	Lg	Md	Md	50	Md	Lg	Md	Md	77	Lg	Lg	Md	Md
24	Sm	Lg	Md	Lg	51	Md	Lg	Md	Lg	78	Lg	Lg	Md	Lg
25	Sm	Lg	Lg	Sm	52	Md	Lg	Lg	Sm	79	Lg	Lg	Lg	Sm
26	Sm	Lg	Lg	Md	53	Md	Lg	Lg	Md	**80**	**Lg**	**Lg**	**Lg**	**Md**
27	Sm	Lg	Lg	Lg	54	Md	Lg	Lg	Lg	**81**	**Lg**	**Lg**	**Lg**	**Lg**

For each problem the DBCP-EA is run 5 times, and *%_Error* is the mean of the % errors for the 5 runs. Table 4-10 shows the maximum, minimum, and average percentage errors (% error), as well as the standard deviation, from the 5 runs for each problem. The information given in this table provides a view of the variability encountered by the DBCP-EA in solving each problem. Solutions with a *%_Error* $\leq$ 1% are deemed good

solutions. Solutions with a *%_Error* > 1% are deemed substandard. Of the 81 problems, 13 had *%_Errors* > 1%. These 13 problems are marked as bold in Table 4-9, and their data are also marked as bold in Table 4-10.

Table 4-10: Maximum % error, Minimum % error, *%_Error*, and Standard Deviation for the 81 Problems

			N = Small				*N* = Medium				*N* = Large			
s	***p***	***d***	**Maximum% error**	**Minimum % error**	***%_Error***	**Standard Deviation**	**Maximum% error**	**Minimum % error**	***%_Error***	**Standard Deviation**	**Maximum% error**	**Minimum % error**	***%_Error***	**Standard Deviation**
Small	Small	Sm	0.00	0.00	0.00	0.00	0.19	0.00	0.07	0.10	1.13	0.00	0.25	0.49
		Md	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		Lg	0.00	0.00	0.00	0.00	0.06	0.06	0.06	0.00	0.12	0.00	0.02	0.05
	Medium	Sm	0.00	0.00	0.00	0.00	0.31	0.18	0.20	0.06	0.29	0.20	0.25	0.04
		Md	0.00	0.00	0.00	0.00	1.18	0.77	0.85	0.19	**2.10**	**1.31**	**1.54**	**0.32**
		Lg	0.00	0.00	0.00	0.00	0.65	0.65	0.65	0.00	**6.45**	**5.41**	**5.62**	**0.46**
	Large	Sm	0.00	0.00	0.00	0.00	0.86	0.00	0.34	0.35	1.29	0.25	0.63	0.43
		Md	0.00	0.00	0.00	0.00	0.79	0.09	0.36	0.29	1.14	0.84	0.99	0.15
		Lg	0.00	0.00	0.00	0.00	**2.29**	**0.98**	**1.24**	**0.58**	**8.55**	**0.94**	**4.44**	**3.86**
Medium	Small	Sm	0.00	0.00	0.00	0.00	0.04	0.00	0.03	0.02	1.07	0.00	0.27	0.46
		Md	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.35	0.20	0.24	0.07
		Lg	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.11	0.11	0.11	0.00
	Medium	Sm	0.00	0.00	0.00	0.00	0.53	0.00	0.21	0.29	**3.63**	**1.12**	**1.66**	**1.10**
		Md	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**2.40**	**2.40**	**2.40**	**0.00**
		Lg	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**1.43**	**1.43**	**1.43**	**0.00**
	Large	Sm	0.00	0.00	0.00	0.00	1.88	0.00	0.38	0.84	0.75	0.26	0.40	0.20
		Md	0.00	0.00	0.00	0.00	**1.57**	**0.67**	**1.18**	**0.47**	**1.68**	**0.90**	**1.08**	**0.34**
		Lg	0.00	0.00	0.00	0.00	**1.39**	**1.05**	**1.12**	**0.15**	**8.47**	**3.83**	**6.03**	**2.10**
Large	Small	Sm	0.00	0.00	0.00	0.00	0.04	0.00	0.02	0.02	0.40	0.27	0.32	0.07
		Md	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.11	0.00	0.06	0.06
		Lg	0.00	0.00	0.00	0.00	0.05	0.00	0.01	0.02	0.00	0.00	0.00	0.00
	Medium	Sm	0.00	0.00	0.00	0.00	0.41	0.00	0.08	0.18	0.96	0.00	0.38	0.52
		Md	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		Lg	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Large	Sm	0.00	0.00	0.00	0.00	0.06	0.06	0.06	0.00	1.00	0.84	0.89	0.07
		Md	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**2.01**	**2.01**	**2.01**	**0.00**
		Lg	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**1.06**	**1.06**	**1.06**	**0.00**

Figure 4-12 shows the *%_Error* plotted for the 81 problems. Ten of the thirteen substandard problem solutions, occur when *N* = Large. Only three substandard problems occur when *N* = Medium and their *%_Errors* are all very close to 1%. Figure 4-13 shows the *%_Error* data for the 27 problems where *N* = Medium and the 27 problems where *N* = Large overlaid on top of each other so that the *%_Error* for problems with the same *s*, *p*, and *d* parameter settings can be compared as the planning horizon increases from 12 periods to 18 periods. The 27 problems where N = Medium correspond to problem number 28-54, and the 27 problems where *N* = Large corresponds to problem number 55 to 81, shown in Table 4-9. Since the *%_Errors* are all zero for the 27 problems were *N* = Small, it is not necessary to plot those points. These two figures show that as the planning horizon increases so does the *%_Error*, and to a lesser extent, in some instances, problems with the same *s*, *p*, and *d* parameter settings follow a similar pattern where the magnitude of the *%_Error* is greatly amplified for the *N* = Large problems.

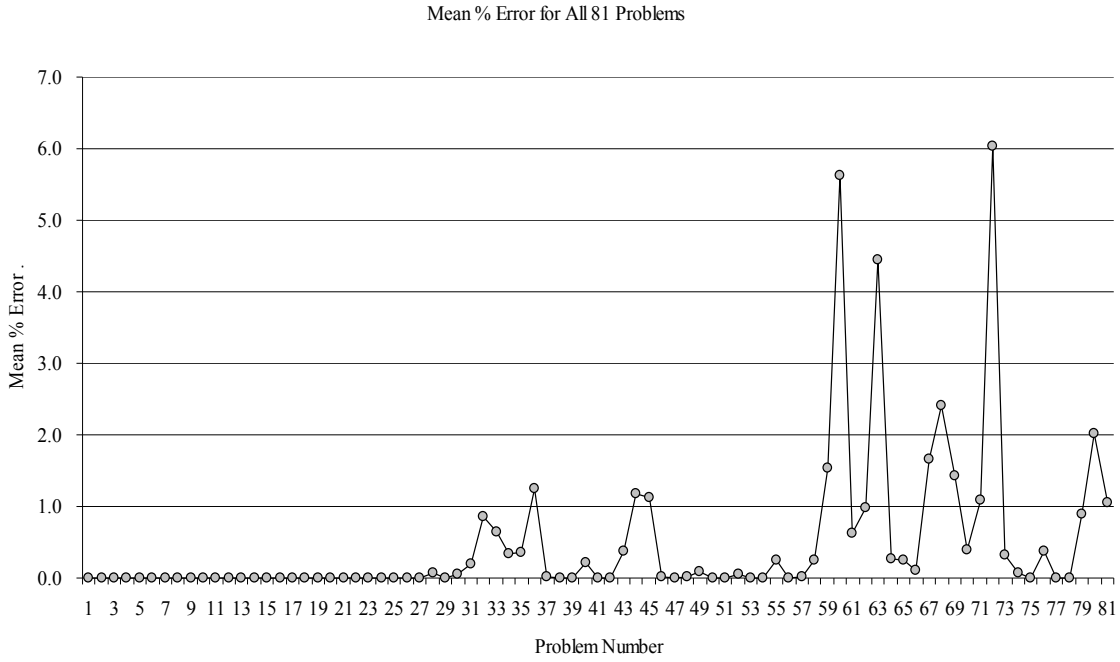
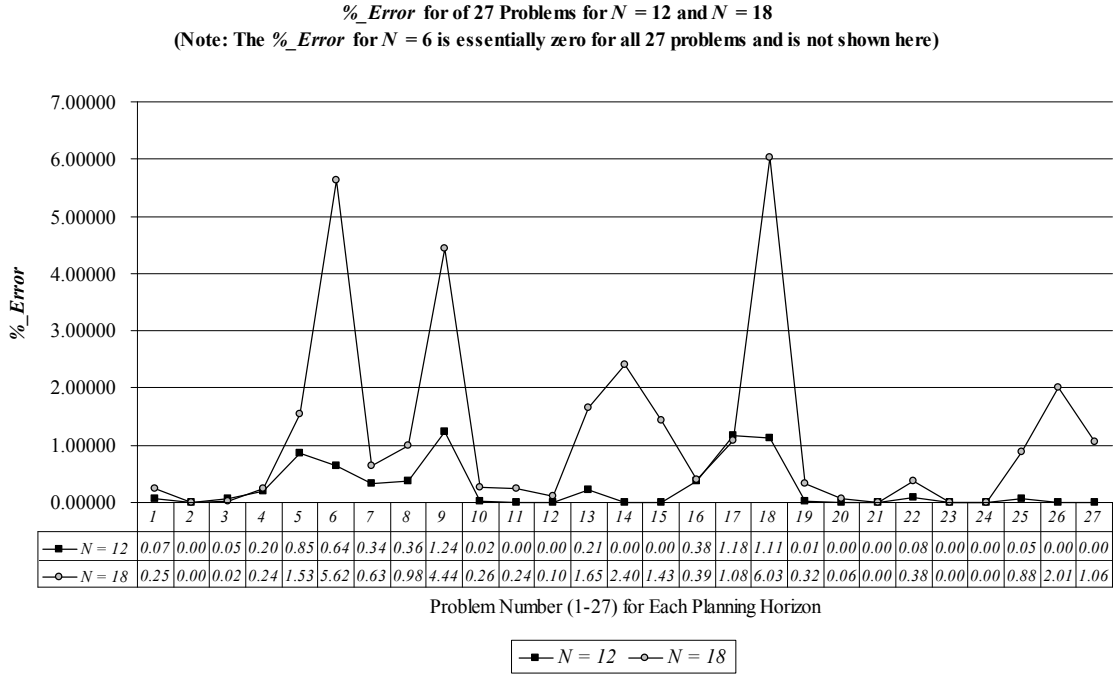


Figure 4-12: *%_Error* for the 81 Problems



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
N = 12	0.07	0.00	0.05	0.20	0.85	0.64	0.34	0.36	1.24	0.02	0.00	0.00	0.21	0.00	0.00	0.38	1.18	1.11	0.01	0.00	0.00	0.08	0.00	0.00	0.05	0.00	0.00
N = 18	0.25	0.00	0.02	0.24	1.53	5.62	0.63	0.98	4.44	0.26	0.24	0.10	1.65	2.40	1.43	0.39	1.08	6.03	0.32	0.06	0.00	0.38	0.00	0.00	0.88	2.01	1.06

Figure 4-13: Overlay of the *%_Errors* for *N* = Medium and Large

Figure 4-14 shows a plot of the standard deviations for the 81 problems listed in Table 4-10. Figure 4-15 shows the standard deviation for the *N* = Medium and *N* = Large

data overlaid on top of each other. Of the 54 standard deviations in Figure 4-15 only 4 (the 9th, 13th, 16th, and 18th problem from the *N* = Large data series) have a standard deviation greater than 0.5. Of these 4 problems, the 9th, 13th, and 18th (from the *N* = Large data series) belonged to substandard solutions. The 18th problem from the *N* = Large data series (problem 72 in Figures 4-12 and 4-14) and 9th problem (problem 63 in Figures 4-12 and 4-14) have the first (*%_Error* = 6.03, standard deviation = 2.09) and third (*%_Errors* = 4.44, standard deviation = 3.86) largest substandard solutions. Interestingly, the second largest *%_Error* (*%_Error* = 5.62) occurred for the 6th problem (problem 60 in Figures 4-12 and 4-14) from the *N* = Large data series and had an average standard deviation of only 0.46. Therefore, while a large standard deviation seems to be correlated with larger *%_Errors*, it is not a required condition. Only two of the 81 problems had a standard deviation greater than 1.1, which means that the DBCP-EA performed consistently.

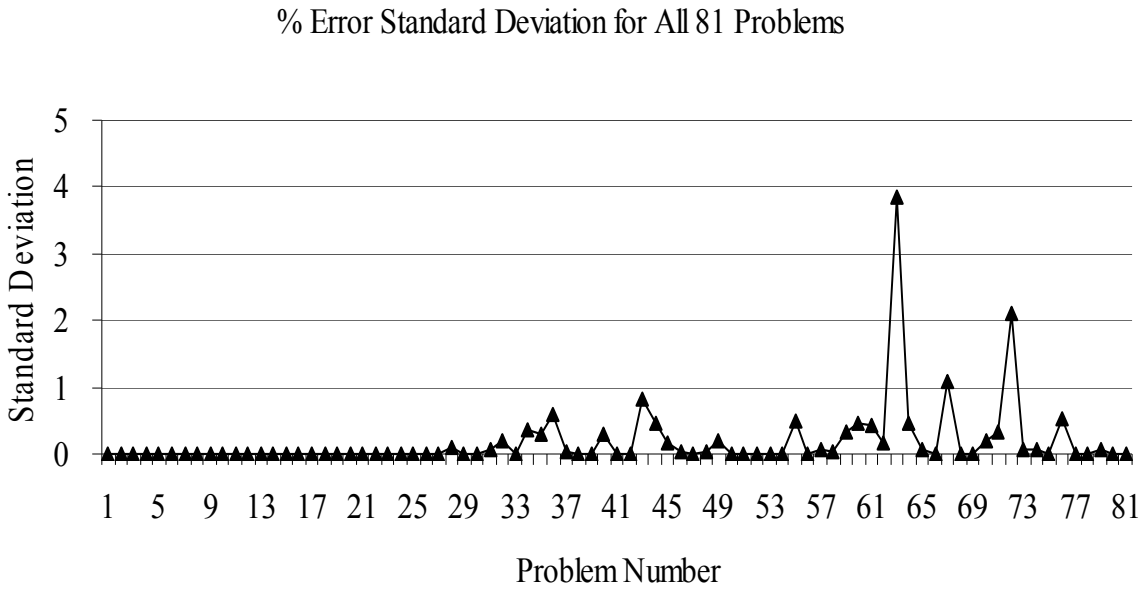
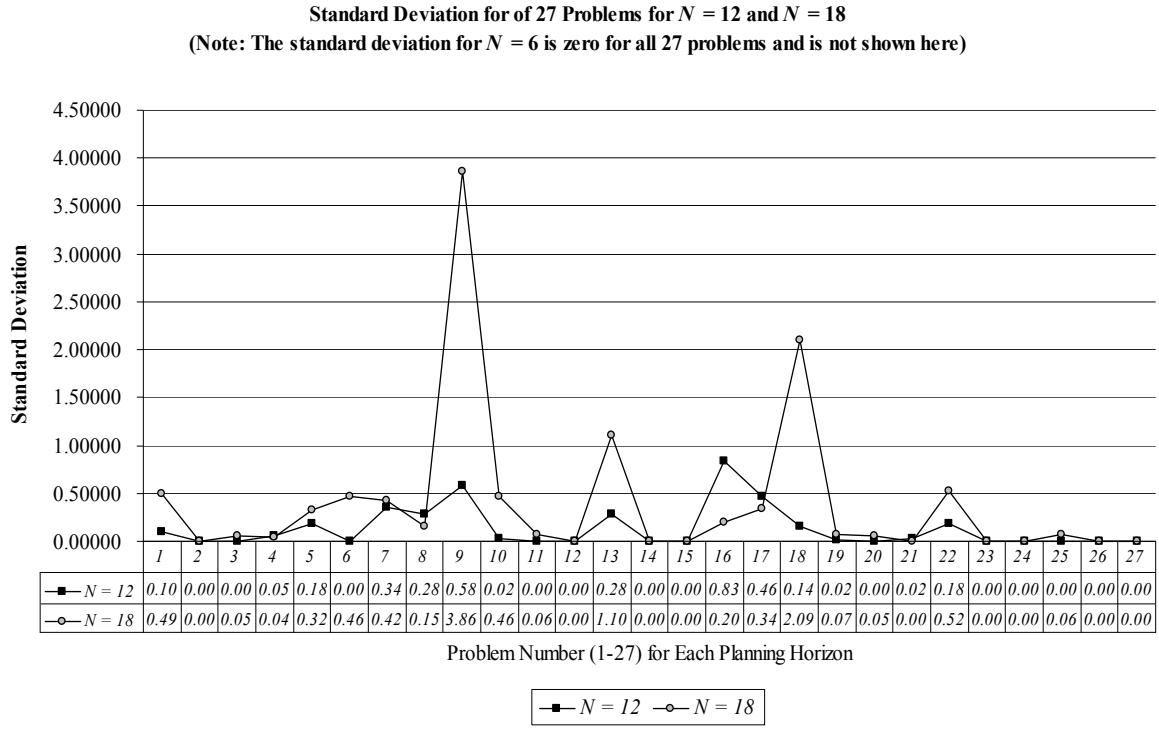


Figure 4-14: *%_Error* Standard Deviation for the 81 Problems



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
N = 12	0.10	0.00	0.00	0.05	0.18	0.00	0.34	0.28	0.58	0.02	0.00	0.00	0.28	0.00	0.00	0.83	0.46	0.14	0.02	0.00	0.02	0.18	0.00	0.00	0.00	0.00	0.00
N = 18	0.49	0.00	0.05	0.04	0.32	0.46	0.42	0.15	3.86	0.46	0.06	0.00	1.10	0.00	0.00	0.20	0.34	2.09	0.07	0.05	0.00	0.52	0.00	0.00	0.06	0.00	0.00

Figure 4-15: Overlay of *%_Error* Standard Deviation for *N* = Medium and Large

Table 4-11 shows the minimum, maximum, and average time it took the DBCP-EA to find the best solution, as well as the standard deviation. As expected, as the *N* increases, so does the computational time. Since the standard deviation for all but a few problems is so low, we have some confidence that the DBCP-EA performs in a consistent manner allowing for fairly accurate estimates of computation time.

The computation time for *N* = Small problems is negligible. For *N* = Medium, problem 35 had the longest maximum run time with 11.5 minutes (691.53 seconds), its best time was 0.4 minutes (25.23 seconds), and its average time for the 5 runs was 4.2 minutes (252.58 seconds). The *%_Error* for problem 35 is 0.36%, with a *%_Error* standard deviation of 0.29. *N* = Large problems have the most variable computational time (see Table 4-11) due to the time required to search a much larger search space. For these problems the longest maximum run time is 49.3 minutes (2,958.48 seconds), which

was for problem 62. The best computational time among the 5 runs for problem 62 was 19.8 minutes (1,186.52 seconds); the average computational time is 36.3 minutes (2,176.49) with a standard deviation of 12.5 minutes. The *%_Error* for problem 62 is 0.99% with a *%_Error* standard deviation of 0.15, the accuracy is good and the variability is low. Therefore, it would be safe to say that the DBCP-EA is fairly consistent at finding good solutions for this problem.

Table 4-11: Maximum, Minimum, and Average EA Computational Time (sec) for the 81 Problems

			N = Small				*N* = Medium				*N* = Large			
s	***p***	***d***	**Maximum**	**Minimum**	**Average**	**Std. Dev.**	**Maximum**	**Minimum**	**Average**	**Std. Dev.**	**Maximum**	**Minimum**	**Average**	**Std. Dev.**
Small	Small	Small	0.38	0.17	0.23	0.1	150.05	9.11	46.34	58.6	1,715.42	95.33	518.68	680.0
		Medium	0.08	0.06	0.07	0.0	41.31	1.31	16.90	17.3	549.67	166.30	300.24	149.3
		Large	0.08	0.06	0.07	0.0	0.23	0.23	0.23	0.0	192.56	0.63	39.02	85.8
	Medium	Small	0.06	0.06	0.06	0.0	384.44	148.92	256.63	116.1	2,768.61	477.08	1,124.60	930.8
		Medium	0.08	0.06	0.07	0.0	123.61	0.22	24.90	55.2	2,224.84	0.58	837.07	863.5
		Large	0.08	0.06	0.07	0.0	0.24	0.22	0.23	0.0	947.92	0.64	243.50	410.4
	Large	Small	0.08	0.06	0.07	0.0	545.19	51.63	210.47	205.9	1,338.56	561.06	932.88	355.2
		Medium	0.08	0.06	0.07	0.0	**691.53**	**25.23**	**252.58**	**269.2**	**2,958.48**	**1,186.52**	**2,176.49**	**750.9**
		Large	0.08	0.06	0.07	0.0	143.69	3.83	77.74	64.7	2,562.67	1,326.42	2,064.77	575.8
Medium	Small	Small	0.48	0.13	0.27	0.2	61.69	13.09	31.85	18.7	1,427.97	224.91	947.08	456.4
		Medium	0.08	0.06	0.07	0.0	20.63	5.27	10.54	6.2	303.14	123.45	198.90	73.7
		Large	0.08	0.06	0.07	0.0	98.33	0.22	19.85	43.9	0.66	0.64	0.65	0.0
	Medium	Small	0.08	0.06	0.07	0.0	236.30	5.89	70.70	96.0	1,502.52	476.94	1,171.90	404.6
		Medium	0.08	0.06	0.07	0.0	0.22	0.20	0.21	0.0	251.80	0.58	134.44	124.0
		Large	0.09	0.06	0.08	0.0	0.25	0.23	0.24	0.0	0.66	0.64	0.64	0.0
	Large	Small	0.08	0.06	0.07	0.0	443.83	58.23	227.05	167.5	1,938.50	190.45	1,287.55	702.4
		Medium	0.08	0.06	0.07	0.0	217.20	58.20	147.65	73.7	2,352.64	681.47	1,402.32	686.6
		Large	0.08	0.06	0.07	0.0	110.98	0.23	22.39	49.5	2,546.69	0.63	936.06	1,289.2

Table 4-11: continued

			N = Small				*N* = Medium				*N* = Large			
s	***p***	***d***	**Maximum**	**Minimum**	**Average**	**Std. Dev.**	**Maximum**	**Minimum**	**Average**	**Std. Dev.**	**Maximum**	**Minimum**	**Average**	**Std. Dev.**
Large	Small	Small	0.38	0.13	0.19	0.1	76.03	4.34	25.49	29.1	2,670.13	144.23	1,097.74	1,036.1
		Medium	0.08	0.06	0.07	0.0	18.28	2.09	7.63	7.0	979.05	7.63	342.68	373.1
		Large	0.08	0.06	0.07	0.0	233.50	0.24	83.35	114.7	999.69	0.64	260.56	433.2
	Medium	Small	0.08	0.06	0.07	0.0	177.39	2.49	49.23	72.6	450.31	59.72	267.78	141.2
		Medium	0.08	0.06	0.08	0.0	0.22	0.22	0.22	0.0	980.05	0.58	428.54	428.9
		Large	0.08	0.06	0.07	0.0	0.25	0.23	0.24	0.0	0.66	0.63	0.64	0.0
	Large	Small	0.08	0.06	0.07	0.0	192.44	21.88	78.32	68.3	2,659.64	1,555.73	2,083.27	445.6
		Medium	0.08	0.06	0.07	0.0	0.23	0.20	0.22	0.0	0.59	0.56	0.58	0.0
		Large	0.08	0.06	0.07	0.0	0.25	0.23	0.24	0.0	0.66	0.63	0.64	0.0

In order to compare the DBCP-EA computational time to that of the MILP the times are overlaid in Figure 4-16. From this figure it is clear that the greatest difference between the computational times for the MILP and the DBCP-EA occur during the N = Large problems. Figures 4-17 through 4-19 compare the computational time of the MILP and the DBCP-EA for N = Small, Medium, and Large problems, respectively.

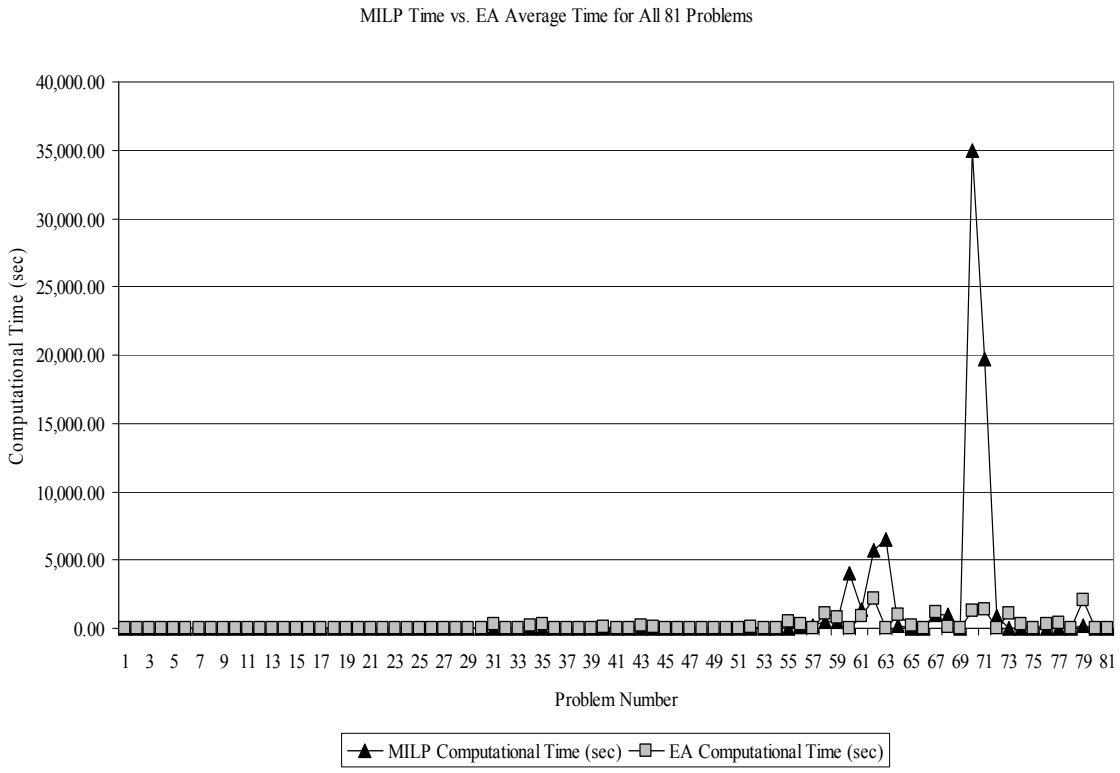


Figure 4-16: EA Computational Average Time (sec) the 81 Problems

Figure 4.17 shows that although the DBCP-EA, on average, took longer to solve N = Small problems, even for the worse cases the times were under 0.5 minutes, which makes it competitive with the MILP formulation. The computational time for the DBCP-EA jumps significantly over that of the MILP for most of the N = Medium problems, as can be seen in Figure 4-18. In most cases, for N = Medium the DBCP-EA found

solutions in less than 1 minute. With respect to the average computational time for any *N* = Medium problem, the worse case was 4.3 minutes (256.63 seconds) for problem 31.

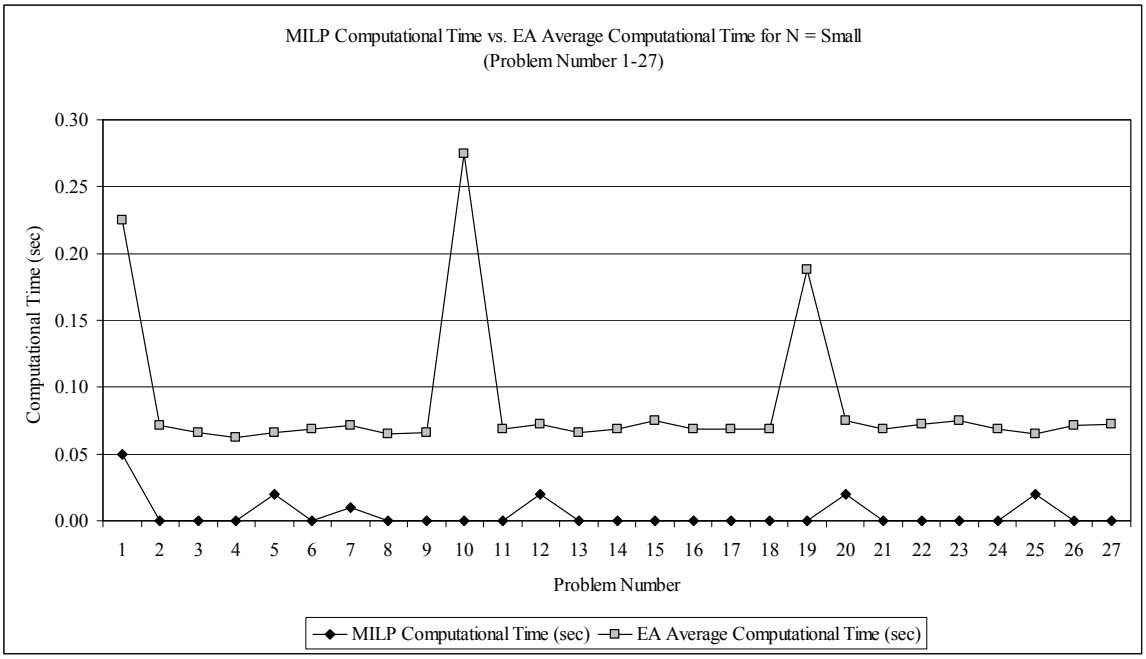


Figure 4-17: Computational Time for MILP and DBCP-EA for *N* = Small

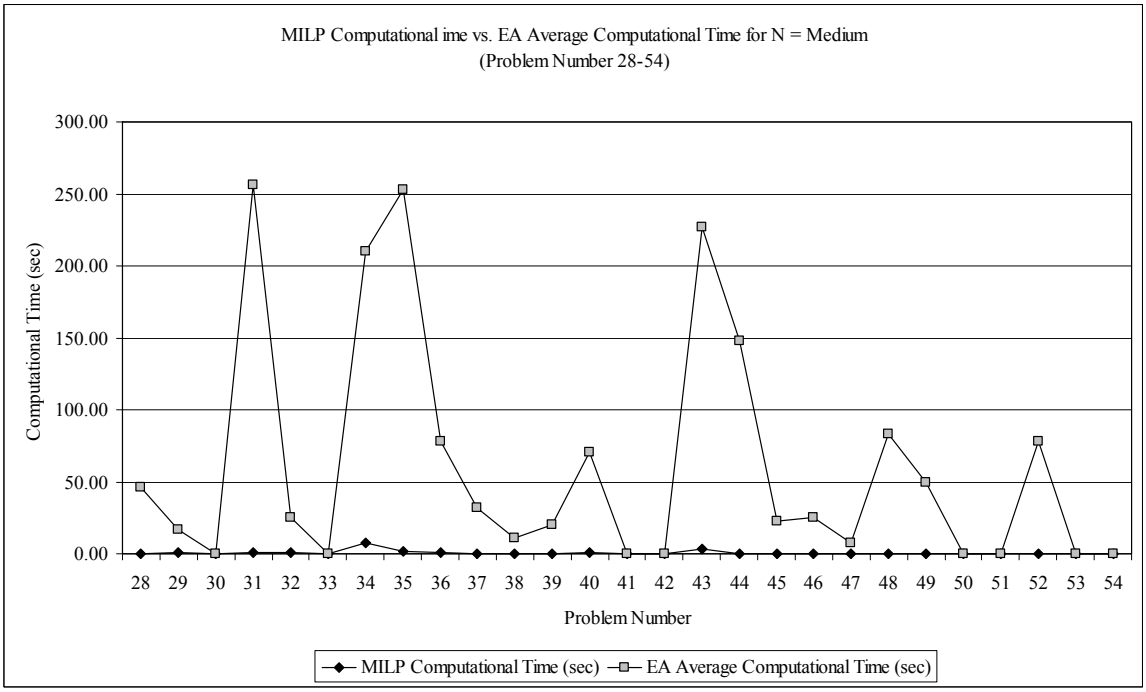


Figure 4-18: Computational Time for MILP and DBCP-EA for *N* = Medium

Figure 4-19 show that the DBCP-EA outperforms the MILP with respect to computational time for 11 out of the 27 N = Large problems. Of these, there are 5 problems where the MILP computational time is clearly larger than the average DBCP-EA computational time; the most significant difference occurs for problem 70. Of the remaining 16 problems the computational times between the DBCP-EA and the MILP are very competitive. For the three planning horizons, the average DBCP-EA computational times were 0.09 seconds, 1 minute, and 11.6 minutes for N = Small, Medium, and Large problems, respectively.

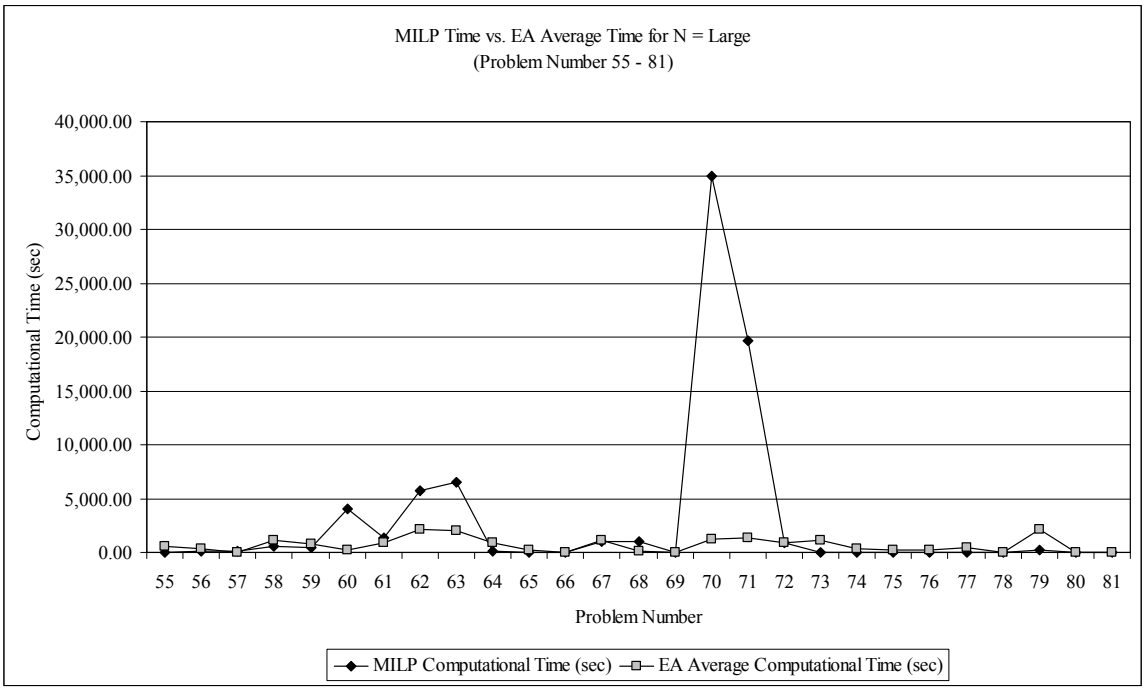


Figure 4-19: Computational Time for MILP and DBCP-EA for N = Large

4.4.1 Examination of DBCP-EA Problems with Substandard Solutions

By examining the 13 DBCP-EA substandard problem solutions insight can be gained into the how the DBCP-EA performs and some common characteristics that exist among these substandard solutions can be discussed.

Table 4-12 shows the % error for the 5 runs for each of the substandard problems. While a majority of the problems (8 out of 13) have *%_Errors* less than 2%, there are 3 problems with *%_Errors* greater than 4%. Interestingly, for some problems, such as problem 81 (also problem 68, 69, and 80), the % error for all 5 runs is the same indicating that while the DBCP-EA was unable to identify a good solution the DBCP-EA did perform consistently. These problems are highlighted in the table.

Table 4-12: DBCP-EA Substandard Problem Solution Data

Problem	**Parameter**				**% error for each Runs**					***%_Error***	**Standard**
Number	***N***	***s***	***P***	***d***	**1**	**2**	**3**	**4**	**5**	**(Avg.)**	**Deviation**
81	Lg	Lg	Lg	Lg	1.06	1.06	1.06	1.06	1.06	1.06	0.00
71	Lg	Md	Lg	Md	0.90	0.90	0.90	1.03	1.68	1.08	0.34
45	Md	Md	Lg	Lg	1.05	1.05	1.05	1.05	1.39	1.12	0.15
44	Md	Md	Lg	Md	0.67	0.67	1.42	1.57	1.57	1.18	0.47
36	Md	Sm	Lg	Lg	0.98	0.98	0.98	0.98	2.29	1.24	0.58
69	Lg	Md	Md	Lg	1.43	1.43	1.43	1.43	1.43	1.43	0.00
59	Lg	Sm	Md	Md	1.31	1.31	1.49	1.49	2.10	1.54	0.32
67	Lg	Md	Md	Sm	1.21	1.21	1.12	1.12	3.63	1.66	1.10
80	Lg	Lg	Lg	Md	2.01	2.01	2.01	2.01	2.01	2.01	0.00
68	Lg	Md	Md	Md	2.40	2.40	2.40	2.40	2.40	2.40	0.00
63	Lg	Sm	Lg	Lg	0.94	0.94	3.25	8.55	8.55	4.44	3.86
60	Lg	Sm	Md	Lg	5.41	5.41	5.41	5.41	6.45	5.62	0.46
72	Lg	Md	Lg	Lg	3.83	3.83	7.02	7.02	8.47	6.03	2.10

Problem 36 has a % error of 0.98 for 4 of the 5 runs; it is only the 5th run with a % error of 2.29 which causes the *%_Error* to be above 1%. This may indicate that the % error for the 5th run might have been an outlier and not representative of the DBCP-EA ability to produce good solutions for this problem. Also, problem 63 two out of the 5 runs had relatively low % errors (both were 0.94 % error) and 2 out of 5 % errors were very large (both were 8.55 % error). The DBCP-EA is inconsistent for this problem

(standard deviation = 3.86) which makes it difficult for the DBCP-EA to find a good solution sometimes. Problem 72 has not only the largest *%_Error*, but also a large standard deviation. So, the % errors for all 5 runs are high as well as being not consistent (%_Error = 6.03, standard deviation = 2.10). On the other hand, the % errors for problem 60 are consistently large but without much variability.

Problems 60, 63, and 72 are three substandard problems that have some unusually large % errors within their 5 runs. These problems all have N = Large and d = Large. For two of these problems s = Small, and for two of the problems p = Large. These three problems also have the largest *%_Errors*.

A possible explanation for these results is that as s increases from Small to Large, contracts with longer duration become optimal because as the temporal discount increases it becomes more attractive to buy long duration contracts. However, when p increases from Small to Large, it is more optimal to buy contracts with smaller durations as price decrease makes contracts with long durations costly. The algorithm's repair function which biases solutions toward contracts with the longest durations works well if long duration contracts are favorable. For N = Small and Medium these biases are not that problematic because the search space was small enough to allows the solution to still find adequate solutions, in most cases. However, for N = Large problems, the search space becomes much bigger and these biases towards the extreme become expensive, which is evidenced in problems 60, 63 and 72. Therefore when N = Large and s = Small and/or p = Large then long duration contracts become expensive. In addition, if d = Large, these long duration contracts becomes even more expensive since these contracts have larger *bws* due to the rapid increase of demand over time. The settings of these

three problems have a bias towards common contracts which produce substandard solutions.

Table 4-13 shows aggregate data pertaining to the 13 DBCP-EA substandard problems. Problems with N = Large are much more likely to have substandard solutions than N = Small or Medium. For s the majority of substandard problems occur when s = Medium. This indicates that the DBCP-EA is much better at identifying the best solutions for s = Small where it can forgo small temporal discounts in favor of future price discounts. Likewise, for s = Large it can take advantage of temporal discounts which could result in lower per unit cost. With respect to N = Large, large decreases in price and/or increases in demand also have more occurrences of large *%_Errors*. This is expected since the greater the rate of change in price and/or demand the more complex the problem becomes and the more sensitive the solution quality is to erroneous choices made early in the planning horizon.

Table 4-13: Crosstab Count Analysis of DBCP-EA Substandard Problem Solutions for *N* versus *s*, *p*, and *d*

	s			
N	Large	Medium	Small	Grand Total
Large	15.38%	38.46%	23.08%	76.92%
Medium	0.00%	15.38%	7.69%	23.08%
Small	0.00%	0.00%	0.00%	0.00%
Grand Total	15.38%	53.85%	30.77%	100.00%
	p			
N	Large	Medium	Small	Grand Total
Large	38.46%	38.46%	0.00%	76.92%
Medium	23.08%	0.00%	0.00%	23.08%
Small	0.00%	0.00%	0.00%	0.00%
Grand Total	61.54%	38.46%	0.00%	100.00%
	d			
N	Large	Medium	Small	Grand Total
Large	38.46%	30.77%	7.69%	76.92%
Medium	15.38%	7.69%	0.00%	23.08%
Small	0.00%	0.00%	0.00%	0.00%
Grand Total	53.85%	38.46%	7.69%	100.00%

4.4.2 Comparison between MILP and DBCP-EA Substandard Problem Solutions

Tables 4-14 through 4-26 show the cumulative purchased bandwidth, required bandwidth, and surplus for each period for both the MILP and the best DBCP-EA solutions for the 13 substandard problems. Each table includes the problem number, the parameter settings, and the % error for the solution shown, the *% _Error*, and the *% _Error* standard deviation. While % error is the measure of error between the MILP solution total cost and the DBCP-EA solution's total cost shown within each table, *%_Error* is the measure of error between the MILP solution total cost and the average of the DBCP-EA solutions' total costs for the 5 runs. The differences between the MILP and the DBCP-EA solutions are highlighted in grey. In problems 36 (Table 4-14, *% error* = 0.98, *%_Error* = 1.24), 45 (Table 4-16, *% error* = 1.05, *%_Error* = 1.12), 60

(Table 4-18, *% error* = 5.41, *%_Error* = 5.62), 68 (Table 4-21, *% error* = 2.40, *%_Error* = 2.40), 69 (Table 4-22, *% error* = 1.43, *%_Error* = 1.43), 71 (Table 4-23, *% error* = 0.90, *%_Error* = 1.08), 72 (Table 4-24, *% error* = 3.83, *%_Error* = 6.03), 80 (Table 4-25, *% error* = 2.01, *%_Error* = 2.01) and 81 (Table 4-26, *% error* = 1.06, *%_Error* = 1.06), the DBCP-EA erroneously chose the longest possible contract in at least one period. This is undoubtedly a bias imposed by the repair function. Future research would attempt to reduce or eliminate this bias.

In problems 59 (Table 4-17, *% error* = 1.31, *%_Error* = 1.54), 63 (Table 4-19, *% error* = 0.94, *%_Error* = 4.44), and 67 (Table 4-20, *% error* = 1.12, *%_Error* = 1.66) the DBCP-EA purchased some contracts that are too short and some contracts that are too long. This type of error could indicate that the DBCP-EA was stopped prematurely. Future research could look at the effects on solution quality and computational time when the EA is allowed to run for more generations. Problem 44 (Table 4-15, *% error* = 0.67, *%_Error* = 1.18) is the exception to the observation that the DBCP-EA tends to purchase contracts that are of longer duration than is necessary. In problem 44 the MILP solution purchases the longest contract available in all periods, whereas the DBCP-EA purchased one contract that was not the longest available. Again, this error might indicate that the DBCP-EA was terminated too soon.

Another interesting observation is that in most cases the MILP and the DBCP-EA solutions purchased only one contract in each period, even though no restriction was imposed. Problem 67, Table 4-20, is the only problem of the 13 substandard solutions that does not have N number of contracts in the optimal solution, where $N = 18$ periods. The MILP optimal solution only purchased 17 contracts; no contracts were purchased

during period 2. The DBCP-EA solution had only 16 contracts. Interestingly, in the optimal solution it is cheaper to purchase a contract with more bandwidth than is needed for period 1, where the amount covered the bandwidth requirement for periods 1 and 2, thereby creating a surplus in period 1. This action is optimal due to the unique parameter settings of N = Large, s = Medium, p = Medium, and d = Small. In this instance it is cheaper to purchase excessive bandwidth in period 1, than to purchase the exact amount of bandwidth needed in period 1 and then purchase another contract in period 2. For this problem the demand increase between periods is small so the surplus bandwidth in period 1 is small, only 54 Mbps. Starting in period 8 the contracts selected by the MILP and the DBCP-EA are identical, but due to difference in contract lengths purchased in pervious periods the bandwidth sizes for the majority of the contracts from the 8th period to the end of the planning horizon have differing *bws* for the two solutions.

Problem 63, Table 4-19, has an odd occurrence. The optimal solution found by the MILP purchases 18 contracts, while the DBCP-EA solution purchases 19 contracts; the % error is 0.94 and *%_Error* is 4.44. In this case, the DBCP-EA solution has two contracts purchased in the 7th period. This occurrence proves that the algorithm does not restrict the number of contract per period.

Table 4-14: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #36

Prob#	*N*	*s*	*P*	*d*	*% error*	*% Error*	Std. Dev.	Rank #
36	Medium	Small	Large	Large	0.98	1.24	0.58	9

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678
MILP Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678
MILP Solution (i.e., Active Contracts)	[01, 07]	[02, 07]	[03, 09]	[04, 10]	[05, 13]	[06, 13]	[07, 13]	[08, 13]	[09, 13]	[10, 13]	[11, 13]	[12, 13]
MILP *bws*	540	162	211	274	356	462	1,304	782	1,227	1,596	1,718	2,233
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
EA *bws*	540	162	211	274	356	462	1,304	782	1,016	1,322	1,718	2,233
EA Best Solution (i.e., Active Contracts)	[01, 07]	[02, 07]	[03, 13]	[04, 13]	[05, 13]	[06, 13]	[07, 13]	[08, 13]	[09, 13]	[10, 13]	[11, 13]	[12, 13]
EA Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-15: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #44

Prob#	*N*	*s*	*P*	*d*	*% error*	*%_Error*	Std. Dev.	Rank #
44	Medium	Medium	Large	Medium	0.67	1.18	0.47	10

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013
MILP Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013
MILP Solution (i.e., Active Contracts)	[01, 13]	[02, 13]	[03, 13]	[04, 13]	[05, 13]	[06, 13]	[07, 13]	[08, 13]	[09, 13]	[10, 13]	[11, 13]	[12, 13]
MILP *bws*	540	108	130	156	186	224	269	322	387	465	557	669
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
EA *bws*	540	108	130	156	186	224	269	322	927	465	557	669
EA Best Solution (i.e., Active Contracts)	[01, 09]	[02, 13]	[03, 13]	[04, 13]	[05, 13]	[06, 13]	[07, 13]	[08, 13]	[09, 13]	[10, 13]	[11, 13]	[12, 13]
EA Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-16: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #45

Prob#	*N*	*s*	*p*	*d*	*% error*	*%_Error*	Std. Dev.	Rank #
45	Medium	Medium	Large	Large	1.05	1.12	0.15	11

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678
MILP Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678
MILP Solution (i.e., Active Contracts)	[01, 09]	[02, 09]	[03, 10]	[04, 13]	[05, 13]	[06, 13]	[07, 13]	[08, 13]	[09, 13]	[10, 13]	[11, 13]	[12, 13]
MILP *bws*	540	162	211	274	356	462	602	782	1,718	1,533	1,718	2,233
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
EA *bws*	540	162	211	274	356	462	602	782	1,178	1,322	1,718	2,233
EA Best Solution (i.e., Active Contracts)	[01, 13]	[02, 09]	[03, 13]	[04, 13]	[05, 13]	[06, 13]	[07, 13]	[08, 13]	[09, 13]	[10, 13]	[11, 13]	[12, 13]
EA Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-17: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #59

Prob#	*N*	*s*	*p*	*d*	*% error*	*% _Error*	Std. Dev.	Rank #
59	Large	Small	Medium	Medium	1.31	1.54	0.32	7

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
MILP Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
MILP Solution (i.e., Active Contracts)	[01, 09]	[02, 09]	[03, 10]	[04, 11]	[05, 12]	[06, 13]	[07, 14]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	108	130	156	186	224	269	322	1,035	595	713	855	1,026	1,232	1,156	1,386	1,664	1,997
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	108	130	156	186	224	269	560	927	465	713	669	802	963	1,156	1,386	1,664	1,997
EA Best Solution (i.e., Active Contracts)	[01, 09]	[02, 08]	[03, 08]	[04, 11]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-18: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #60

Prob#	*N*	*s*	*P*	*d*	*% error*	*% Error*	Std. Dev.	Rank #
60	Large	Small	Medium	Large	5.41	5.62	0.46	2

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Solution (i.e., Active Contracts)	[01, 09]	[02, 09]	[03, 11]	[04, 12]	[05, 13]	[06, 14]	[07, 15]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	162	211	274	356	462	602	782	1,718	1,322	1,929	2,507	3,259	4,237	5,508	6,379	8,292	10,780
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	162	211	274	356	462	602	782	1,178	1,322	1,718	2,233	2,903	3,775	4,906	6,379	8,292	10,780
EA Best Solution (i.e., Active	[01, 19]	[02, 09]	[03, 19]	[04, 19]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-19: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #63

Prob#	*N*	*s*	*P*	*d*	*% error*	*% _Error*	Std. Dev.	Rank #
63	Large	Small	Large	Large	0.94	4.44	3.86	3

MILP Surplus	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607		3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607		3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Solution (i.e., Active Contracts)	[01, 07]	[02, 08]	[03, 09]	[04, 10]	[05, 11]	[06, 12]	[07, 13]	—	[08, 14]	[09, 15]	[10, 16]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	162	211	274	356	462	1,142		944	1,227	1,596	2,074	2,695	4,045	4,719	6,133	7,975	8,292	10,780
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***-***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	162	211	274	356	462	195	947	944	1,501	1,322	2,074	2,890	2,903	4,722	4,906	6,379	8,292	10,780
EA Best Solution (i.e., Active Contracts)	[01, 07]	[02, 08]	[03, 09]	[04, 09]	[05, 11]	[06, 12]	[07, 12]	[07, 14]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,200	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
Required Bandwidth	540	702	913	1,187	1,543	2,005		2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
EA Surplus	0	0	0	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0

Table 4-20: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #67

Prob#	*N*	*s*	*P*	*d*	*% error*	*% Error*	Std. Dev.	Rank #
67	Large	Medium	Medium	Small	1.12	1.66	1.10	6

MILP Surplus	**54**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	594	654	719	791	870	957	1,053	1,158	1,274	1,401	1,541	1,695	1,865	2,051	2,256	2,482	2,730
MILP Cumulative *bws*	594	594	654	719	791	870	957	1,053	1,158	1,274	1,401	1,541	1,695	1,865	2,051	2,256	2,482	2,730
MILP Solution (i.e., Active Contracts)	[01, 12]		[03, 10]	[04, 11]	[05, 12]	[06, 13]	[07, 15]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	594		60	65	72	79	87	96	105	176	192	806	233	170	273	205	226	248
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	114		65	151		87	96	105	230	192	140	154	710	186	205	226	248
EA Best Solution (i.e., Active Contracts)	[01, 14]	[02, 10]		[04, 11]	[05, 19]		[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	654	654	719	870	870	957	1,053	1,158	1,274	1,401	1,541	1,695	1,865	2,051	2,256	2,482	2,730
Required Bandwidth	540	594	654	719	791	870	957	1,053	1,158	1,274	1,401	1,541	1,695	1,865	2,051	2,256	2,482	2,730
EA Surplus	0	**60**	0	0	**79**	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-21: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #68

Prob#	*N*	*s*	*P*	*d*	*% error*	*% Error*	Std. Dev.	Rank #
68	Large	Medium	Medium	Medium	2.40	2.40	0.00	4

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
MILP Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
MILP Solution (i.e., Active Contracts)	[01, 12]	[02, 10]	[03, 12]	[04, 13]	[05, 14]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	108	130	156	186	224	269	322	387	573	557	1,339	958	1,149	1,156	1,386	1,664	1,997
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	108	130	156	186	224	269	322	387	573	557	669	802	963	1,156	1,386	1,664	1,997
EA Best Solution (i.e., Active Contracts)	[01, 19]	[02, 10]	[03, 19]	[04, 19]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-22: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #69

Prob#	*N*	*s*	*P*	*d*	*% error*	*% _Error*	Std. Dev.	Rank #
69	Large	Medium	Medium	Large	1.43	1.43	0.00	8

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Solution (i.e., Active Contracts)	[01, 12]	[02, 11]	[03, 13]	[04, 14]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	162	211	274	356	462	602	782	1,016	1,322	1,880	2,773	3,114	4,049	4,906	6,379	8,292	10,780
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	162	211	274	356	462	602	782	1,016	1,322	1,718	2,233	2,903	3,775	4,906	6,379	8,292	10,780
EA Best Solution (i.e., Active Contracts)	[01, 19]	[02, 19]	[03, 19]	[04, 19]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-23: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #71

Prob#	*N*	*s*	*P*	*d*	*% error*	*% Error*	Std. Dev.	Rank #
71	Large	Medium	Large	Medium	0.90	1.08	0.34	12

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
MILP Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
MILP Solution (i.e., Active Contracts)	[01, 09]	[02, 09]	[03, 10]	[04, 11]	[05, 12]	[06, 13]	[07, 15]	[08, 16]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	108	130	156	186	224	269	322	1,035	595	713	855	1,026	963	1,425	1,708	1,664	1,997
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	108	130	156	186	224	269	322	1,035	595	713	669	802	963	1,156	1,386	1,664	1,997
EA Best Solution (i.e., Active Contracts)	[01, 09]	[02, 09]	[03, 10]	[04, 11]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-24: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #72

Prob#	*N*	*s*	*P*	*d*	*% error*	*% _Error*	Std. Dev.	Rank #
72	Large	Medium	Large	Large	3.83	6.03	2.10	1

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Solution (i.e., Active Contracts)	[01, 09]	[02, 09]	[03, 10]	[04, 12]	[05, 13]	[06, 14]	[07, 15]	[08, 16]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	162	211	274	356	462	602	782	1,718	1,533	1,718	2,507	3,259	4,237	5,508	7,161	8,292	10,780
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	162	211	274	356	462	602	782	1,556	1,322	1,718	2,233	2,903	3,775	4,906	6,379	8,292	10,780
EA Best Solution (i.e., Active Contracts)	[01, 09]	[02, 19]	[03, 19]	[04, 19]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-25: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #80

Prob#	*N*	*s*	*P*	*d*	*% error*	*% Error*	Std. Dev.	Rank #
80	Large	Large	Large	Medium	2.01	2.01	0.00	5

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
MILP Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
MILP Solution (i.e., Active Contracts)	[01, 13]	[02, 10]	[03, 12]	[04, 13]	[05, 15]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	108	130	156	186	224	269	322	387	573	557	799	1,498	963	1,342	1,386	1,664	1,997
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	108	130	156	186	224	269	322	387	465	557	669	802	963	1,156	1,386	1,664	1,997
EA Best Solution (i.e., Active Contracts)	[01, 19]	[02, 19]	[03, 19]	[04, 19]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
Required Bandwidth	540	648	778	934	1,120	1,344	1,613	1,935	2,322	2,787	3,344	4,013	4,815	5,778	6,934	8,320	9,984	11,981
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4-26: Cumulative Purchased Bandwidth, Required Bandwidth, and Surplus for the MILP and Best DBCP-EA Solutions for Problem #81

Prob#	*N*	*s*	*P*	*d*	*% error*	*% Error*	Std. Dev.	Rank #
81	Large	Large	Large	Large	1.06	1.06	0.00	13

MILP Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
MILP Solution (i.e., Active Contracts)	[01, 13]	[02, 11]	[03, 13]	[04, 15]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
MILP *bws*	540	162	211	274	356	462	602	782	1,016	1,322	1,880	2,233	3,654	3,775	5,180	6,379	8,292	10,780
Period	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
EA *bws*	540	162	211	274	356	462	602	782	1,016	1,322	1,718	2,233	2,903	3,775	4,906	6,379	8,292	10,780
EA Best Solution (i.e., Active Contracts)	[01, 19]	[02, 19]	[03, 19]	[04, 19]	[05, 19]	[06, 19]	[07, 19]	[08, 19]	[09, 19]	[10, 19]	[11, 19]	[12, 19]	[13, 19]	[14, 19]	[15, 19]	[16, 19]	[17, 19]	[18, 19]
EA Cumulative *bws*	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
Required Bandwidth	540	702	913	1,187	1,543	2,005	2,607	3,389	4,405	5,727	7,445	9,678	12,581	16,356	21,262	27,641	35,933	46,713
EA Surplus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

4.4.3 Aggregate Data Analysis for all DBCP-EA Problem Solutions

Table 4-27 shows another aggregated view for all 81 of the DBCP-EA problem solutions. *MOM_%_Error* is used to denote the mean *%_Error*. For all two dimensional comparisons where *N* a parameter, the largest errors only occur when *N* = Large. Longer planning horizons are inherently more difficult to solve than smaller problems, therefore is observation is not surprising. Comparing the *MOM_%_Error* with respect to *N* versus *s* the *%_Error* are large only when *N* = Large and *s* = Small and Medium. With *N* versus *p*, large errors occur when *N* = Large and *p* = Medium and Large. For *N* versus *d,* large errors occur when *N* and *d* are both Large. With *s* versus *p*, *s* = Small and Medium and *p* = Large produced a large error, and, with respect to *s* versus *d,* the only large error occurs when *s* = Small and *d* = Large. For *p* versus *d*, the only large error occurs when both *p* and *d* are Large.

The DBCP-EA appears to have trouble identifying a good solution when there is a choice between a moderate duration discount (i.e., *s* = Medium) and a rapid rate of decrease in price over time (*p* = Large). The DBCP-EA does not wait to purchase bandwidth until the last possible moment where the advantage of lower prices can be realized, but instead opts to purchase longer duration contracts. Likewise, when the duration discount is small (*s* = Small) and there is a rapid increase in demand over time (*d* = Large) the DBCP-EA has difficulty identifying good solutions. The DBCP-EA does not exploit the long duration contracts when the duration discount is small, thereby incurring higher costs when demand rapidly increases for subsequent periods in the planning horizon. These types of erroneous decisions made early in the planning process may not be obvious until late the planning horizon. Looking at *p* versus *d,* the only large

error occurs when *p* and *d* are both Large. This is because when *p* = Large it is typically prudent to wait to purchase bandwidth as late as possible in order to take advantage of reduced costs. However, DBCP-EA's repair function biases solutions towards long term contracts and when *d* = Large, these long term contracts become even more expensive as more bandwidth is bought for long term contracts due to large increases in demand.

Table 4-27: Two-Dimensional Analysis for Aggregate Data for *N* vs. *s*, *N* vs. *p*, *N* vs. *d*, *s* vs. *p*, *s* vs. *d*, *p* vs. *d*

			***MOM_%_Error* (Std. Dev.)**			
			***s* = Small**	***s* = Medium**	***s* = Large**	***Average***
N* vs. *s	***N***	**Small**	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
		Medium	0.42 (0.17)	0.32 (0.20)	0.02 (0.03)	0.26 (0.13)
		Large	1.53 (0.65)	1.52 (0.48)	0.53 (0.08)	1.19 (0.40)
			***p* = Small**	***p* = Medium**	***p* = Large**	***Average***
N* vs. *p	***N***	**Small**	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
		Medium	0.02 (0.19)	0.22 (0.08)	0.52 (0.30)	0.26 (0.13)
		Large	0.14 (0.13)	1.48 (0.27)	1.95 (0.80)	1.19 (0.40)
			***d* = Small**	***d* = Medium**	***d* = Large**	***Average***
N* vs. *d	***N***	**Small**	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
		Medium	0.16 (0.21)	0.27 (0.11)	0.34 (0.08)	0.26 (0.13)
		Large	0.56 (0.38)	0.93 (0.11)	2.08 (0.72)	1.19 (0.40)
			***p* = Small**	***p* = Medium**	***p* = Large**	***Average***
s* vs. *p	***s***	**Small**	0.05 (0.07)	1.01 (0.12)	0.89 (0.63)	0.65 (0.27)
		Medium	0.07 (0.06)	0.64 (0.15)	1.13 (0.46)	0.61 (0.22)
		Large	0.05 (0.02)	0.05 (0.08)	0.45 (0.01)	0.18 (0.04)
			***d* = Small**	***d* = Medium**	***d* = Large**	***Average***
s* vs. *d	***s***	**Small**	0.20 (0.16)	0.42 (0.11)	1.34 (0.55)	0.65 (0.27)
		Medium	0.33 (0.32)	0.55 (0.10)	0.97 (0.25)	0.61 (0.22)
		Large	0.19 (0.10)	0.23 (0.01)	0.12 (0.00)	0.18 (0.04)
			***d* = Small**	***d* = Medium**	***d* = Large**	***Average***
p* vs. *d	***p***	**Small**	0.11 (0.13)	0.03 (0.01)	0.02 (0.01)	0.05 (0.05)
		Medium	0.31 (0.24)	0.53 (0.06)	0.86 (0.05)	0.57 (0.12)
		Large	0.30 (0.21)	0.63 (0.14)	1.55 (0.74)	0.82 (0.36)

Figures 4-20 through 4-22 show *MOM_%_Error* versus *s* and three values of *p* for *N* = Small, Medium, and Large, respectively. As can be seen when *N* = Small (Figure

4-20), the error is extremely small and there is little, if any impact of *s* or *p* on the quality of the solution. When *N* = Medium (Figure 4-21), we can see that when *p* = Small, *MOM_%_Error* is close to zero. This is because small price decreases favor solutions with contract of long duration and DBCP-EA's repair function biases the algorithm towards such contracts. When *p* = Medium and Large, we can see that *MOM_%_Error* decline as *s* increases. This is because large discounts overshadow the price decreases again favoring contracts with longer duration. However, when discounts are small, contracts of shorter duration are favored with larger price decreases. While DBCP-EA's second repair function considers contracts with the shortest possible period (period of one), there is no search towards contracts of intermediate time periods. So when *d* is larger it causes DBCP-EA to settle into suboptimal solutions which is shown with a larger *MOM_%_Error*. A similar phenomenon is observed when *N* = Large (Figure 4-22). In this case however the *MOM_%_Error* is much larger due to the greater complexity of the search space when *N* = Large.

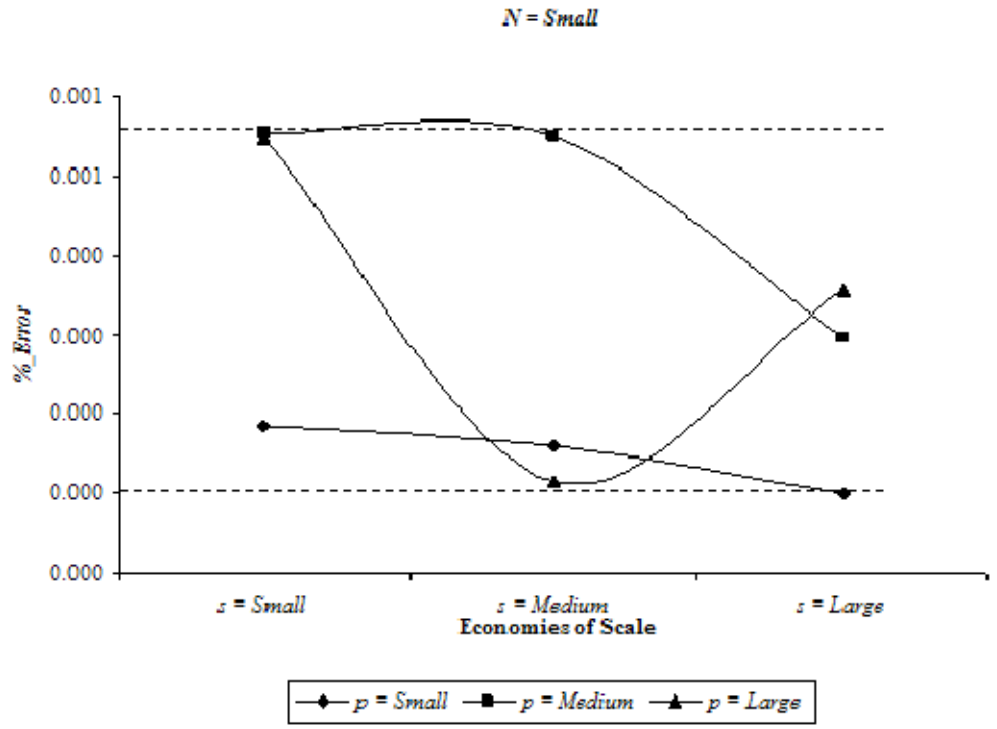


Figure 4-20: Percent Error versus Temporal Discount (s) for N = Small and Three Values of Price Decrease Rate (p)

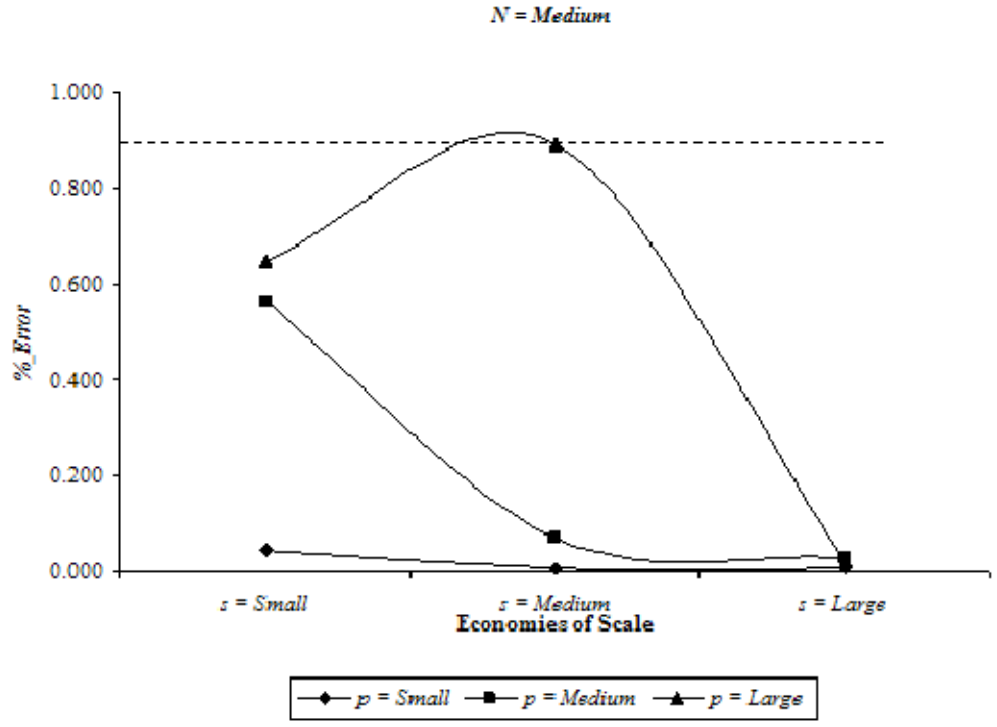


Figure 4-21: Percent Error versus Temporal Discount (s) for N = Medium and Three Values of Price Decrease Rate (p)

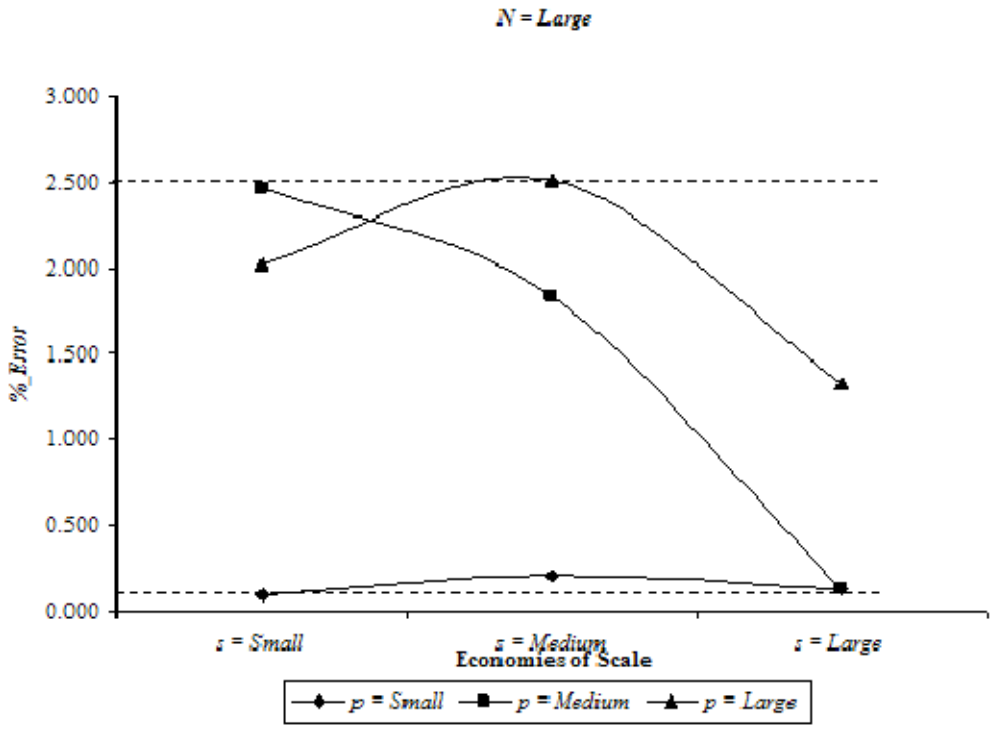


Figure 4-22: Percent Error versus Temporal Discount (*s*) for *N* = Large and Three Values of Price Decrease Rate (*p*)

Figures 4-23 through 4-25 show *MOM_%_Error* versus *s* and three values of *p* for *N* = Small, Medium, and Large, respectively. When *N* = Small (Figure 4-23), the *MOM_%_Error* is very small and DBCP-EA finds the optimal or close to optimal solutions for all combinations of *s* and *d*. When *N* = Medium (Figure 4-24) or Large (Figure 4-25), we can see that as *s* increases the *MOM_%_Error*. When *d* = Large we see that *MOM_%_Error* is the largest when *s* = Small because it is typically prudent to wait to purchase bandwidth as late as possible in order to take advantage of reduced costs. However, DBCP-EA's repair function biases solutions towards long term contracts and when *d* = Large, these long term contracts become even more expensive as more bandwidth is bought for long term contracts due to large increases in demand.

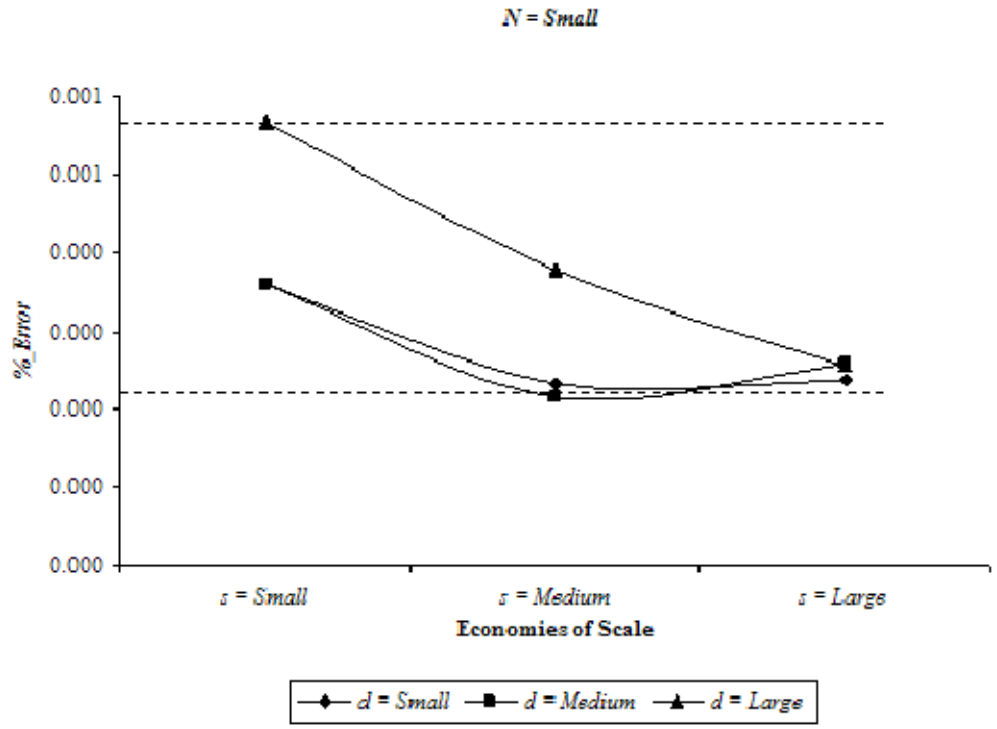


Figure 4-23: Percent Error versus Temporal Discount (*s*) for *N* = Small and Three Values for Demand Increase Rate (*d*)

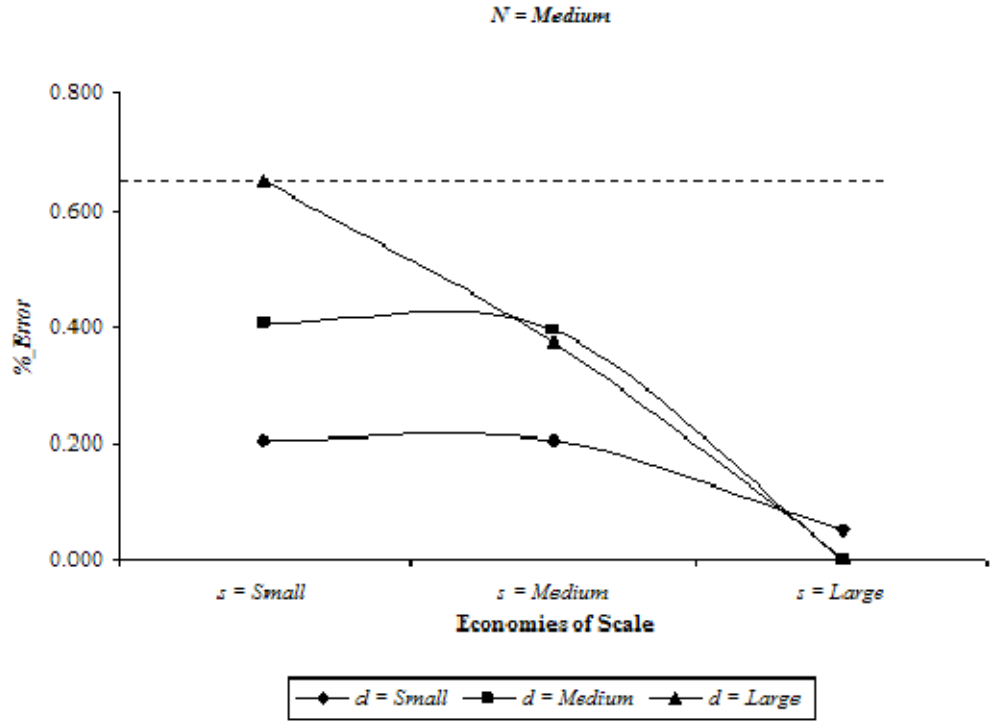


Figure 4-24: Percent Error versus Temporal Discount (*s*) for *N* = Medium with respect to varying Demand Increase Rate (*d*)

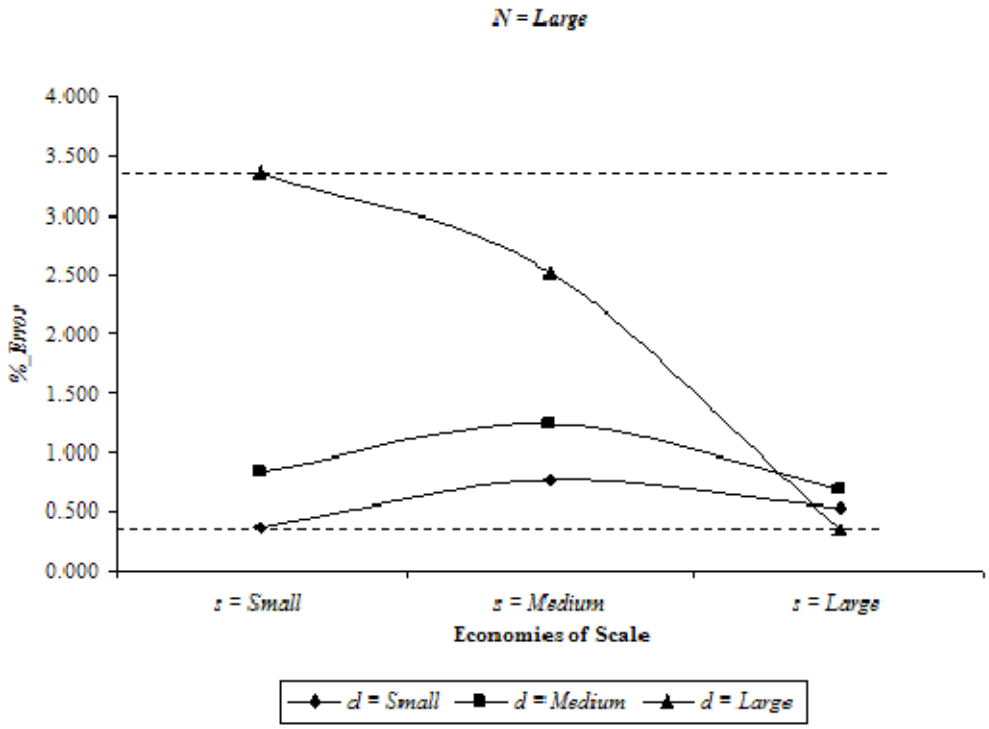


Figure 4-25: Percent Error versus Temporal Discount (*s*) for *N* = Large and Three Values for Demand Increase Rate (*d*)

Figures 4-26 through 4-28 show *MOM_%_Error* versus *d* and three values of *p* for *N* = Small, Medium, and Large, respectively. When *N* = Small (Figure 4-26), the *MOM_%_Error* is very small or close to zero. When *N* = Medium (Figure 4-27) or Large (Figure 4-28) we see similar behavior except when *N* = Large the *MOM_%_Error* is much larger to due to the increased complexity in the search space. As *p* increases from Small to Large, *MOM_%_Error* also increases. This is because as *p* increases, long term contracts become more expensive and DBCP-EA which is biased towards long term contracts due to its repair functions find more expensive solutions. It also clear that this effect is magnified when *d* = Large as the long term contract have a larger quantities making them much more expensive.

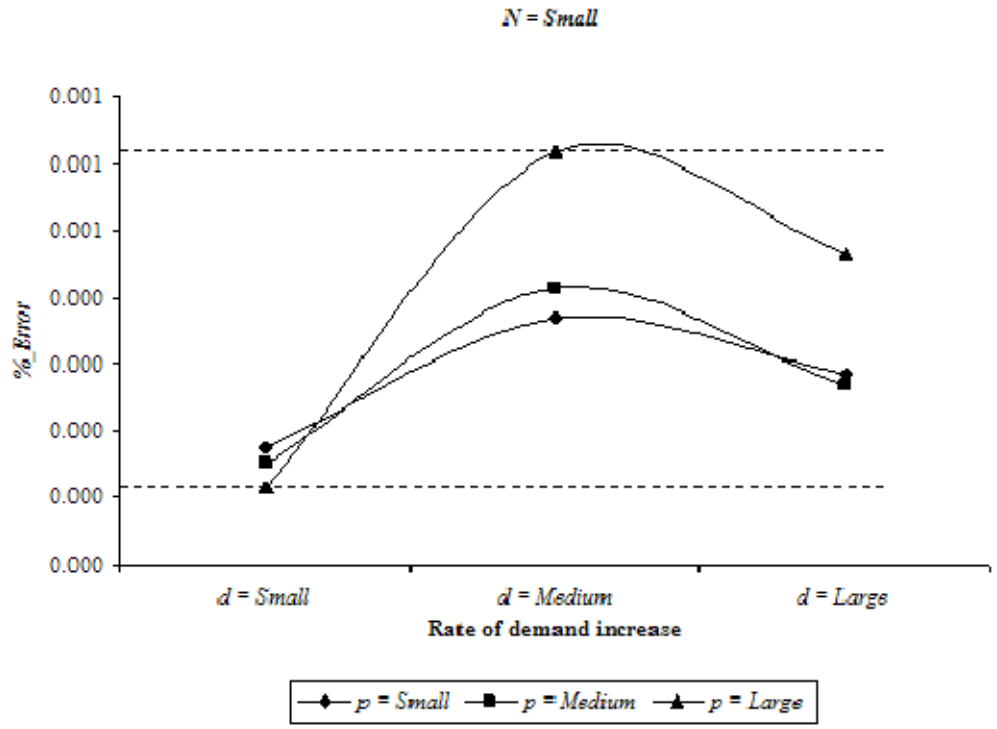


Figure 4-26: Percent Error versus Demand Increase Rate (*d*) for *N* = Small and Three Values for Price Decrease Rate (*p*)

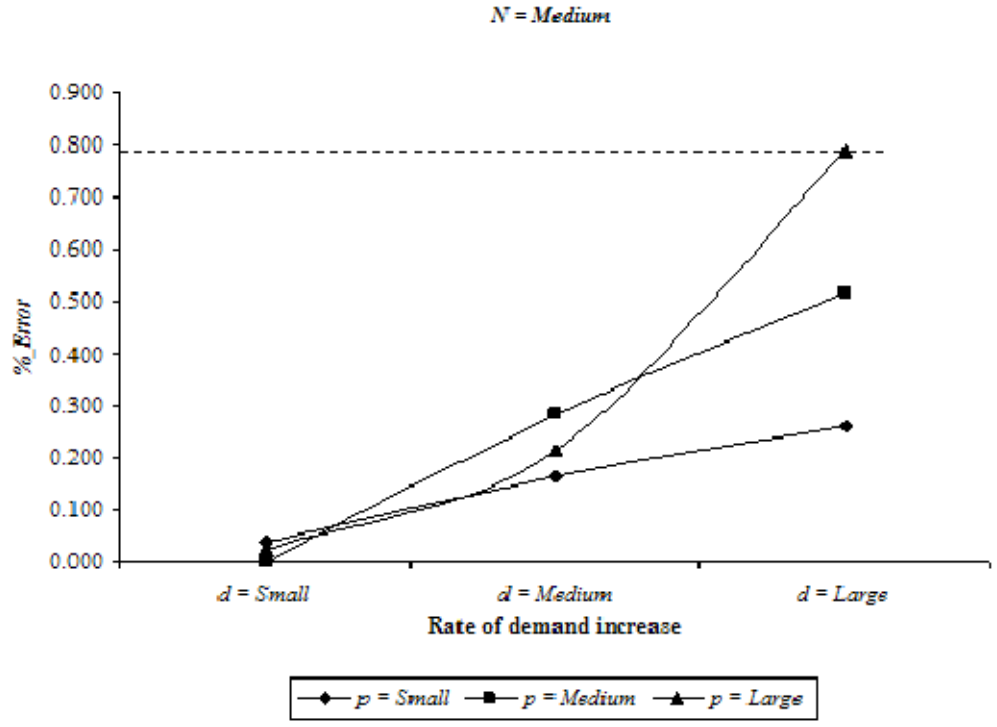


Figure 4-27: Percent Error versus Demand Increase Rate (*d*) for *N* = Medium and Three Values for Price Decrease Rate (*p*)

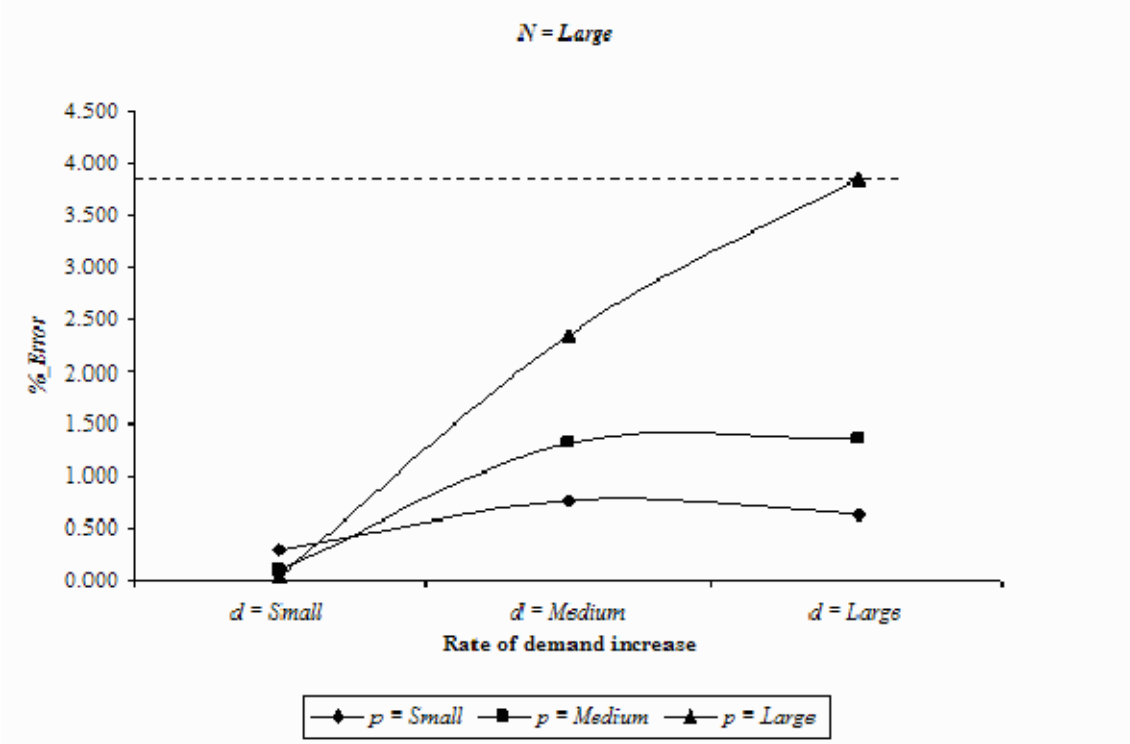


Figure 4-28: Percent Error versus Demand Increase Rate (d) for N = Large and Three Values for Price Decrease Rate (p)

4.5 Final Comments

The DBCP-EA algorithm is able to find optimal or close to optimal solutions (>99% of optimal) for a majority of the problems (68 out of 81). Out of the 13 of the 81 problems that DBCP-EA had substandard solutions only 5 out of 13 had a solution quality less than 98% of the optimal.

Solution quality for DBCP-EA is affected by planning horizon. The DBCP-EA performs very well for short planning horizons. The solution quality is progressively poorer for longer planning horizons. This is due to the fact that the search space explodes as N increases. For example, the number of possible contracts for an N = Small periods is 21, for N = Medium the number of possible contracts is 78, and for N = Large the number of possible contracts is 171 (see Table 2-2). This translates into countless possible

combinations of contracts that can be purchased to satisfy a problem. The assertion that the search space for larger *N* problems is very large is confirmed when the number of iterations performed by CPLEX are examined. In fact, CPLEX was unable to find an optimal integer solution for *N* = Very Large (i.e., 24 period) problems in a reasonable amount of time.

DBCP-EA is also affected by the values of *s* , *p* and *d*. This is due to the nature of the repair functions used. The repair function, though not intentionally designed to do so, exhibits some influence on the quality of solutions and the diversity of the chromosomes. There are two repair algorithms within the repair functions; however these repair algorithms are biased. The first repair algorithm identifies the periods with deficit bandwidth. It then identifies the most common contract, and then the deficit amount is added into that contract. Feasibility is rechecked. If the chromosome still has periods with deficit bandwidth amounts the procedure is repeated. The procedure repeats until feasibility is achieved. The second repair algorithm identifies the periods with bandwidth deficits, and then adds the deficit amount into the first contract that could be purchased in that period (i.e., a one month contract beginning in the deficit period). This is a greedy procedure.

When *s* increases from Small to Large, solutions with contracts with longer duration will have lower cost, when *p* increases from Small to Large, solutions with contracts with shorter duration will have lower costs. When these parameters oppose each other it is difficult for the DBCP-EA to identify an optimal solution based on the influence exhibited by the two repair algorithms within the repair function. It may be that under this scenario mid-range duration contract purchases would be optimal. This is the

kind of purchasing strategy we see in many of the optimal MILP solutions. For the longer horizon problems, typically the beginning periods will have contracts that are active for only a few periods. As time moves on, longer duration contracts are purchased. This kind of behavior is not encouraged by the two repair algorithms within the repair function. Future research could examine the inclusion of different repair algorithms within the repair function.

CHAPTER 5 THE STOCHASTIC BANDWIDTH CONTRACT PROBLEM EVOLUTIONARY ALGORITHM (SBCP-EA)

A major source of risk for organizations is the risk of bandwidth price and demand fluctuations. The BCP-EA presented in Chapter 4 assumes a deterministic demand increase rate and price decrease rate. The stochastic bandwidth contract problem (SBCP) is a modification of the DBCP-EA which employees a simulation model with the evolutionary algorithm that incorporates stochastic demand and price for bandwidth purchases. Previous research has incorporated simulations within EAs to solve problems within stochastic environments (Azadivar, and Wang, 2000; Paris, and Pierreval, 2001; Pierreval, and Tautou, 1997). The simulation is used to generate problem instances drawn from uniform demand and price probability distributions. A group of problem instances are used to evolve and evaluate solutions during the run of the SBCP-EA. Another group of problem instances are used to re-evaluate these same solutions. The objective is to identify solutions that perform well under a wide range of problem instances; such solutions are said to be robust.

5.1 The DBCP-EA versus the SBCP-EA

The DBCP-EA, discussed in Chapter 4, was modified in two ways for the SBCP-EA. The first modification to the DBCP-EA was to the termination condition. The SBCP-EA termination condition has been simplified from that used in the DBCP-EA, which had more than one termination criterion. For the SBCP-EA the maximum number

of generations was set to 500. This value was selected based on the results of preliminary tests. The purpose of restricting the number of generations is to limit the computational time of the SBCP-EA while still maintaining good results.

The second modification to the DBCP-EA was to the representation of the price and demand parameters. In the SBCP-EA the rate of change in price and demand are probabilistic. Therefore the stochastic parameters are the probabilities of a change in price (Δp) and a change in demand (Δd). Both Δp and Δd have three settings: Large, Medium, and Small. The uniform distribution where the probability of a Large change is drawn randomly from a uniform probability distribution ranging from 0.05 to 0.15. The Medium probability distribution range is from 0.07 to 0.13, and the Small probability distribution range is from 0.09 to 0.11. All three probability distribution ranges have an expected value of 0.1, which allows the effects of the different spreads of the range of values to be compared. Since all three ranges have the same expected value we are on average using the deterministic problem where $p = 0.1$ and $d = 0.1$ with a stochastic component added.

The two remaining parameters are temporal discount (s) and planning horizon (N). Table 5-1 show the four parameters and the three measures for each which are designated as Small, Medium, and Large. These designations are used throughout the remainder of this chapter. The *Pop_Size* remains constant at 40 chromosomes throughout the run of the SBCP-EA.

Table 5-1: Size Designations for the Three Values for each of the Four Parameters: *N*, *s*, *Δd*, and *Δp*

Parameters	Small	Medium	Large
N	6	12	18
s	1	2	3
Δp	0.09-0.11	0.07-0.13	0.05-0.15
Δd	0.09-0.11	0.07-0.13	0.05-0.15

5.2 The Three Parts of the SBCP-EA

Unlike the DBCP-EA, the SBCP-EA has three parts: (1) the training phase, (2) the testing phase to re-evaluate solutions, and (3) testing for robustness. Within the following sections variable labels are used in an effort to simplify the discussion. Variable labels are italicized, and after each label a brief description is given.

5.2.1 The SBCP-EA Training Phase

The training phase of the SBCP-EA is the only time that the stochastic EA (SEA) is run. During the run of the SEA solutions are evolved and evaluated based on a randomly generated set of price and demand instances for each problem.

IN: The number of price-demand instances used to evaluate the fitness value of a solution. *IN* values of 100 and 500 were used. *IN* of 100 indicates that there are 100 instances of price and 100 instances of demand for each period within the planning horizon for a problem. A single instance contains price and demand entries for each period in a problem's planning horizon.

Price-Demand_Array: An array of length *N* and width of (2 * *IN*) (*IN* price and *IN* demand instances). The total number of price entries recorded within a *Price-Demand_Array* would also be (*N* * *IN*), and the total number of demand entries would be (*N* * *IN*). There are two different *Price-*

Demand_Arrays: *Price-Demand_Array_1* and *Price-Demand_Array_2*, both have the same *IN*. For each problem, *Price-Demand_Array_1* is used in the training phase, and *Price-Demand_Array_2* is in the testing phase.

Training_Chromosomes: A two dimensional array populated at the end of the training phase with the 20 best chromosomes found during the training phase for each problem.

Training_fv: A one dimensional array of length 81 which contains the average *fv* of the 20 chromosomes in the *Training_Chromosomes* array calculated for each problem.

5.2.1.1 The Stochastic Parameters: Price and Demand

For the sake of brevity only the price parameter will be discussed. The demand variables can be identified by replacing 'Demand' wherever 'Price' is found in the following discussion.

Prob_Price_Change: The cumulative probability from period to period that price will decrease.

Price_Delta: A randomly generated number indicating a price decrease rate from one period to the next. This number is recorded within one of the *Price-Demand_Arrays*.

The two stochastic parameters price (Δp) and demand (Δd) are procedurally virtually identically. The only difference between them is the direction of change. *Price_Delta* is the percentage reduction in price, while *Demand_Delta* is the percentage increase in price.

5.2.1.2 The Creation of the Price-Demand_Arrays

The two *Price-Demand_Arrays* are created before the SEA is run, and are created with respect to a problem's price and demand distribution ranges, as well as the length of the planning horizon as shown in Table 5-2. For example, if N = Small and Δp = Large, then the price instance part of a price-demand instance in *Price-Demand_Array_1* for SEA1 would contain a price entry for all 6 periods generated based on a uniform distribution within the range of 0.05 to 0.15 if a change in price is selected, else a "1" is recorded if a change in price is not selected. For either of the two *Price-Demand_Arrays* the total number of entries is equal to the number of *Price_Delta* values plus the number of "1"s. Therefore, for SEA1 when IN = 100 there will be (N * 100*)* price entries, whereas for SEA2 when IN = 500 there will be (N * 500*)* price entries.

Table 5-2: Parameter Settings for each of the 81 Stochastic Problems

#	*N*	*s*	*Δp*	*Δd*	#	*N*	*s*	*Δp*	*Δd*	#	*N*	*s*	*Δp*	*Δd*
1	Sm	Sm	Sm	Sm	28	Md	Sm	Sm	Sm	55	Lg	Sm	Sm	Sm
2	Sm	Sm	Sm	Md	29	Md	Sm	Sm	Md	56	Lg	Sm	Sm	Md
3	Sm	Sm	Sm	Lg	30	Md	Sm	Sm	Lg	57	Lg	Sm	Sm	Lg
4	Sm	Sm	Md	Sm	31	Md	Sm	Md	Sm	58	Lg	Sm	Md	Sm
5	Sm	Sm	Md	Md	32	Md	Sm	Md	Md	59	Lg	Sm	Md	Md
6	Sm	Sm	Md	Lg	33	Md	Sm	Md	Lg	60	Lg	Sm	Md	Lg
7	Sm	Sm	Lg	Sm	34	Md	Sm	Lg	Sm	61	Lg	Sm	Lg	Sm
8	Sm	Sm	Lg	Md	35	Md	Sm	Lg	Md	62	Lg	Sm	Lg	Md
9	Sm	Sm	Lg	Lg	36	Md	Sm	Lg	Lg	63	Lg	Sm	Lg	Lg
10	Sm	Md	Sm	Sm	37	Md	Md	Sm	Sm	64	Lg	Md	Sm	Sm
11	Sm	Md	Sm	Md	38	Md	Md	Sm	Md	65	Lg	Md	Sm	Md
12	Sm	Md	Sm	Lg	39	Md	Md	Sm	Lg	66	Lg	Md	Sm	Lg
13	Sm	Md	Md	Sm	40	Md	Md	Md	Sm	67	Lg	Md	Md	Sm
14	Sm	Md	Md	Md	41	Md	Md	Md	Md	68	Lg	Md	Md	Md
15	Sm	Md	Md	Lg	42	Md	Md	Md	Lg	69	Lg	Md	Md	Lg
16	Sm	Md	Lg	Sm	43	Md	Md	Lg	Sm	70	Lg	Md	Lg	Sm
17	Sm	Md	Lg	Md	44	Md	Md	Lg	Md	71	Lg	Md	Lg	Md
18	Sm	Md	Lg	Lg	45	Md	Md	Lg	Lg	72	Lg	Md	Lg	Lg
19	Sm	Lg	Sm	Sm	46	Md	Lg	Sm	Sm	73	Lg	Lg	Sm	Sm
20	Sm	Lg	Sm	Md	47	Md	Lg	Sm	Md	74	Lg	Lg	Sm	Md
21	Sm	Lg	Sm	Lg	48	Md	Lg	Sm	Lg	75	Lg	Lg	Sm	Lg
22	Sm	Lg	Md	Sm	49	Md	Lg	Md	Sm	76	Lg	Lg	Md	Sm
23	Sm	Lg	Md	Md	50	Md	Lg	Md	Md	77	Lg	Lg	Md	Md
24	Sm	Lg	Md	Lg	51	Md	Lg	Md	Lg	78	Lg	Lg	Md	Lg
25	Sm	Lg	Lg	Sm	52	Md	Lg	Lg	Sm	79	Lg	Lg	Lg	Sm
26	Sm	Lg	Lg	Md	53	Md	Lg	Lg	Md	80	Lg	Lg	Lg	Md
27	Sm	Lg	Lg	Lg	54	Md	Lg	Lg	Lg	81	Lg	Lg	Lg	Lg

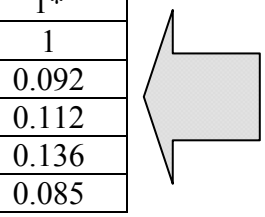
5.2.1.3 Generation of Price Entries into the Price-Demand_Arrays

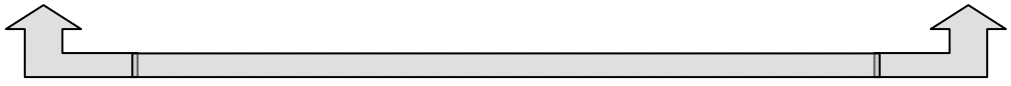
There are two steps to generating the price entries for each period within each instance. The procedure is as follows: (1) Determine if a price decrease should occur in the next period. (2a) If yes, how much should price decrease. (2b) Else, the price for the next period will remain the same as the previous.

The first step requires a randomly generated number (*Rand*) between 0 and 1 to determine if a change should occur. *Rand* is compared to *Prob_Price_Change*, a cutoff variable initially set to 0.50. The second step is to record a price entry within a *Price-Demand_Array*. If *Rand* ≤ *Prob_Price_Change*, then a *Price_Delta* value will be randomly generated based on a uniform distribution from the Δp distribution range specified for that problem. These ranges are shown in Table 5.1. If *Rand* > *Prob_Price_Change* then the price remains unchanged from the previous period, a price entry of "1" is recorded for that period within that instance, and *Prob_Price_Change* is increased by 0.05., thereby providing a slightly higher probability that a price change will occur in the next period. Table 5-3 shows an example of how one price instance might be calculated for a problem with 6 periods (i.e., N = Small), where Δp = Large.

Table 5-3: Example for Calculating One Instance of Price Data for Problem 9: N = Small where Δp = Large
*Period 1 always contains "1" which means that the initial price is the same for all instances

Period Number	**Instance #1**	***Rand*** (a randomly generated number between 0 and 1)	***Prob_Price_Change***	***Rand*** ≤ ***Prob_Price_Change***	***Price_Delta***
1	1*	N/A	N/A	N/A	N/A
2	1	0.83	0.50	False (No Change)	N/A
3	0.092	0.25	0.55	True	0.092
4	0.112	0.39	0.55	True	0.112
5	0.136	0.11	0.55	True	0.136
6	0.085	0.52	0.55	True	0.085





When populating a *Price-Demand_Array* the cells in the first row of all instances contain a price entry of 1, or 100%. Therefore, the first period for all instances has a price change value of '1,' indicating that the same initial price is used for all instances, for all problems. The cell in the first column (i.e., the first instance) and the second row represents the price entry for the second period in the first instance. This cell will contain a *Price_Delta* value drawn from the distribution range for Δp if a change has been selected, else a price entry of "1" is recorded. For each period with in each instance a price entry is recorded. This price entry is be multiplied by the actual price value used within the SEA to make purchasing decisions for a given period, and gives the actual price value to be used to make purchasing decisions for the next period. Table 5-4 shows an example of 8 price-demand instances that could be found in either of the *Price-Demand_Arrays* for a problem 9 which has 6 periods.

Table 5-4: Example of Price and Demand Entries for Problem 9: N = Small, s = Small, Δp = Large, and Δd = Large

	Price-Demand Instances															
N	**1**		**2**		**3**		**4**		**5**		**6**		**7**		**8**	
	Price	Demand	Price	Demand	Price	Demand	Price	Demand	Price	Demand	Price	Demand	Price	Demand	Price	Demand
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	0.069	1	0.061	1	1	1.091	0.125	1	0.120	1	0.120	1	1	1
3	0.012	1.147	0.076	1.086	0.118	1.094	1	1.131	0.118	1.091	0.115	1.050	0.127	1	0.068	1.144
4	0.148	1	0.075	1	1	1.065	1	1.119	0.066	1.055	0.108	1	0.056	1.083	0.062	1.113
5	0.072	1.131	0.125	1	0.060	1	0.062	1.062	0.124	1.088	0.071	1.070	0.141	1.122	0.134	1.122
6	0.135	1.075	0.116	1.101	0.104	1	0.143	1.144	0.065	1	0.143	1	0.061	1.068	0.075	1.107

5.2.1.4 Procedure for Calculating the Training_fv

During the run of an SEA a population of chromosomes evolves. During this evolutionary process the chromosomes are continuously evaluated using *Price-*

Demand_Array_1 with *IN* price-demand instances to obtain the chromosome's *fv*. Each chromosome's *fv* is calculated *IN* times, once for every instance within *Price-Demand_Array_1*, thereby producing *IN* fitness values. The average of the *IN* fitness values is the *fv* assigned to a chromosome. For each problem, 20 best chromosomes are recorded in the *Training_Chromosomes* array, and the average *fv* for the 20 best chromosomes is recorded in the *Training_fv* array. The *Training_fv* array is used in part three of the SBCP-EA, the test for robustness.

5.2.2 The SBCP-EA Testing Phase

The testing phase of the SBCP-EA takes the 20 best chromosomes identified in the training phase for each problem and re-evaluates them using another set of price-demand instances located in *Price-Demand_Array_2*. As with the training phase, the testing phase introduces a number of variable labels to simplify the discussion. Variable labels are italicized, and after each label a brief description is given.

NTR: The number of training runs, which is the number of times an SEA's solutions are re-evaluated, each time using a newly populated *Price-Demand_Array_2*. The number of training runs is 5 and 1 for SEA1 and SEA2, respectively.

Testing_fv: A two dimensional array of a length equal to *NTR* and width equal to 81, which contains the average fitness value of the 20 chromosomes for each problem recorded in the *Training_Chromosomes* array after the chromosomes have been re-evaluated using *Price-Demand_Array_2*.

The testing phase begins by creating the *Price-Demand_Array_2*. The 20 best chromosomes for each problem found during the training phase are re-evaluated *IN*

times, once for each of the *IN* price-demand instances recorded in *Price-Demand_Array_2*, and then the average fitness values for each problem is recorded in the *Testing_fv* array. SEA1 and SEA2 were both run once in the training phase. However, while SEA2 was only tested once (*NTR* = 1), SEA1 was tested 5 times (*NTR* = 5). Once each of the 20 best chromosomes is re-evaluated using *Price-Demand_Array_2* each of the 20 chromosomes will have a new *fv*. The average of the 20 fv is the *Testing_fv* for each problem. The *Testing_fv* is used in part three of the SBCP-EA, the test for robustness.

5.2.3 The SBCP-EA Test for Robustness Phase

The test for robustness allows for a comparison between the average fitness value using the price-demand instances recorded in *Price-Demand_Array_1* and those recorded in *Price-Demand_Array_2*. Those problems whose solution's fitness values were not very different after being evaluated using both *Price-Demand_Arrays* are viewed as being robust across a variety of price-demand possibilities within a given range. More robust solutions are more desirable than solutions that vary greatly depending on price and demand fluctuations. As in the previous two sections variable labels have been included in this section to simplify the discussion. Variable labels are italicized, and after each label a brief description is given.

%_Error: The mean percent error for a problem calculated by subtracting the *Testing_fv* from the *Training_fv* , dividing the result by the *Training_fv*, and then multiplying by 100,

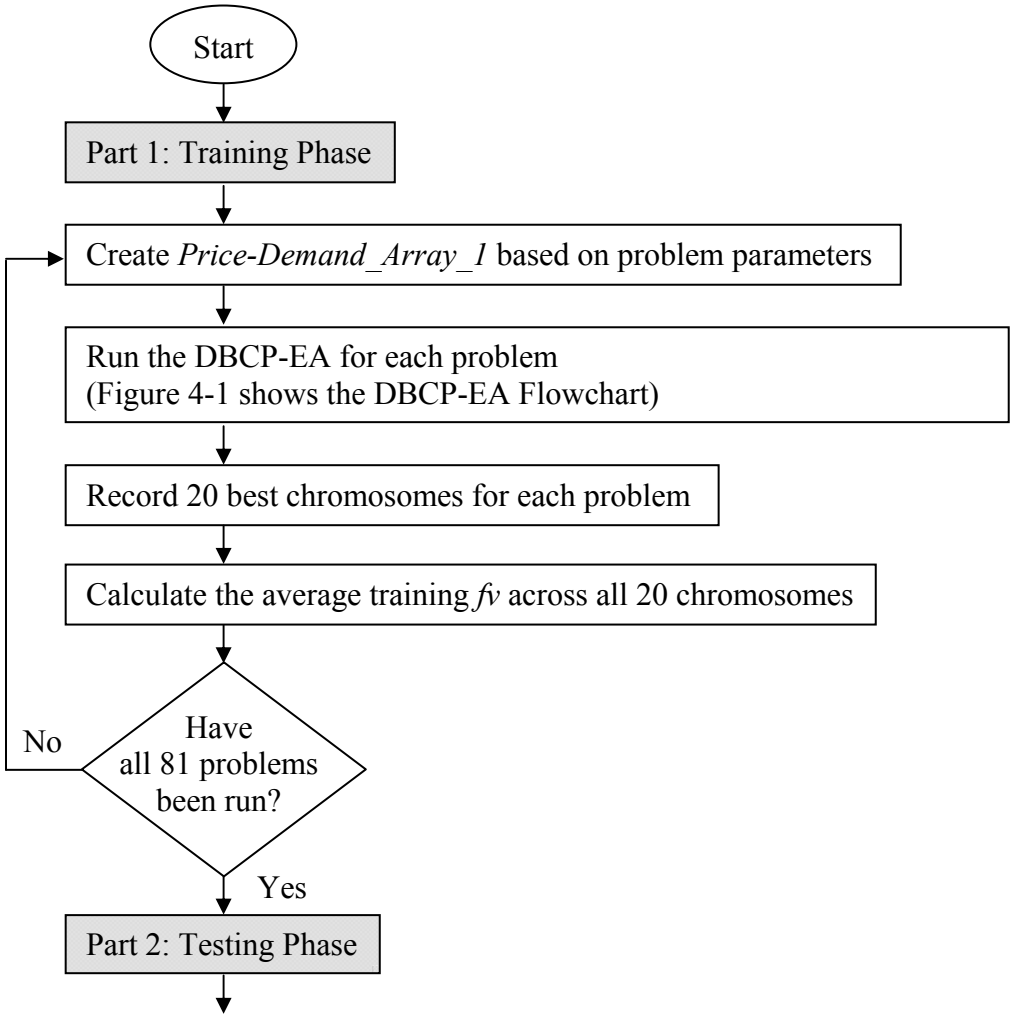
$$i.e., \left(\frac{Training\_fv - Testing\_fv}{Training\_fv} \right) \times 100 = \%\_Error.$$

In Chapter 4 the DBCP-EA solutions were compared to the optimal solutions identified by the MILP, however within a stochastic environment there is no optimal solutions. In this chapter robustness is a surrogate quality measure used to evaluate each problem's solution. The idea is that in a stochastic environment it is best to have a solution(s) for a given problem that will perform well over varying values of the stochastic parameters. Therefore, when comparing the average *fv* produced in the training phase with the average *fv* produced in the testing phase a measure of robustness is found for each problem. Ideally, a problem's solutions should exhibit little variation, or error. Such solutions would be considered robust. Conversely, if the error term is large the problem's solutions would not be considered robust. There is no predetermined 'good' measure of robustness, but this procedure allows us to perform a comparison between problem solutions for different stochastic price and demand values. A test for robustness was run once for the 81 SEA2 problem solutions, whereas the 81 SEA1 problem solutions were tested for robustness five times, each time with a newly populated *Price-Demand_Array_2*.

For all problems, the *Training_fv* will always be less than *Testing_fv* since *Price_Demand_Array_1* was used to create the 20 best solutions for each problem. Therefore, all measures of error will be negative. For SEA2 the measure of error is *%_Error,* the measures of error for SEA1 are *%_Error* and *MOM_%_Error*.

MOM_%_Error: An error term variable that holds the average *%_Error* for the 5 SEA1 testing runs for each of the 81 problems.

Figure 5-1 shows a flowchart for the SBCP-EA, followed by the SBCP-EA pseudo code in Section 5.3. The source code for SBCP-EA is given in Appendix B.


Start
Part 1: Training Phase
Create *Price-Demand_Array_1* based on problem parameters
Run the DBCP-EA for each problem
(Figure 4-1 shows the DBCP-EA Flowchart)
Record 20 best chromosomes for each problem
Calculate the average training *fv* across all 20 chromosomes
Have all 81 problems been run?
No
Yes
Part 2: Testing Phase

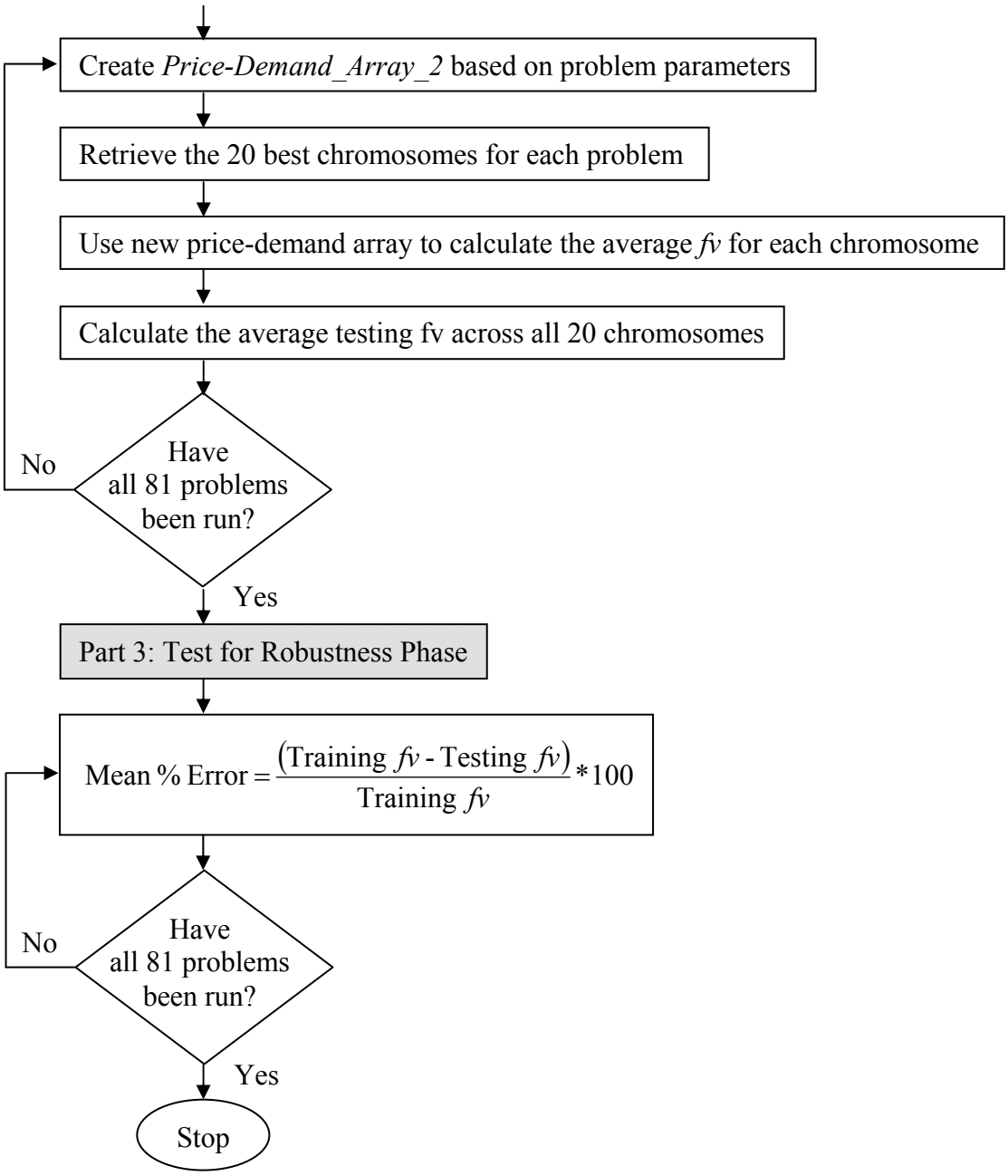


Figure 5-1: SBCP-EA Flowchart

5.3 SBCP-EA Pseudo Code

Within the pseudo-code presented below, an asterisk (i.e., *) is used to marks those steps within the training phase which are unique to SBCP-EA. The testing phase

and test for robustness phase in the SBCP-EA have no corresponding parts in the DBCP-EA.

Part 1: Training Phase Pseudo Code

1. Select *N*.
2. Set *Pop_Size* = 40 chromosomes.
3. Set operator parameters: *One-Point Crossover Operator Rate*, *Uniform Crossover Operator Rate*, *and Mutation Operator Rate.* Note: ($0 \leq$ Operator Rate ≤ 1).
4. *Set the maximum number of generations to run = 500 generations. (See Section 5.1.)
5. Set loop to run 3 times, once for each planning horizon size
    a. *N* = Small
    b. *N* = Medium
    c. *N* = Large
6. *Set *Training_fv* = 0, *Testing_fv* = 0, and *NTR_Count* = 0.
7. Create initial population of chromosomes and place them in the *Chromosome_Array*. Calculate the *fv* for each chromosome and place the value in index 0 of each chromosome vector.
8. Set loop to run 3 times, once for each temporal discount scheme
    a. *s* = 1 for Small temporal discount
    b. *s* = 2 for Medium temporal discount
    c. *s* = 3 for Large temporal discount
9. *Create *Price-Demand_Array_1* which contains 100 price and demand probabilities for each period.
10. Set loop to run 9 times, once for all possible price and demand combination (i.e., 3 x 3 = 9).
    a. Price decrease probability distribution ranges:
        i. Δp = Small range: 0.09-0.11
        ii. Δp = Medium range: 0.07-0.13
        iii. Δp = Large range: 0.05-0.15
    b. Demand increase probability distribution ranges:
        i. Δd = Small range: 0.09-0.11
        ii. Δd = Medium range: 0.07-0.13
        iii. Δd = Large range: 0.05-0.15
11. Copy generation's chromosomes from the *Chromosome_Array* into the *Pool_Array.*
12. Initiate the one-point crossover operator. Append two offspring to the end of the *Pool_Array* each time the one-point crossover operator is implemented.

13. Initiate the uniform crossover operator. Append two offspring to the end of the *Pool_Array* each time the uniform crossover operator is implemented.
14. Initiate the mutation operator. Append one offspring to the end of the *Pool_Array* each time the mutation operator is implemented.
15. Check all chromosomes in the *Pool_Array* for feasibility. If infeasible, then correct using the repair function.
16. *Use the evaluation function to assign a fitness value (i.e., *fv*) to each chromosome in the *Pool_Array*.

    Modification of the DBCP-EA Evaluation Function for the SBCP-EA

    a. Set *Counter* = 0.
    b. *Counter* = *Counter* + 1.
    c. For chromosome number *Counter* do the following:
        i. Evaluate the *fv* 100 times using the 100 price and demand combination data in *Price-Demand_Array_1*.
        ii. Calculate the average *fv* for the 100 *fv* created in previous step.
        iii. Record the average *fv* calculated in previous step into index 0 of the chromosome.
        iv. If *Counter* < *Pop_Size*, then go to step 16 b. Else, done!
17. Select the next generation's chromosomes from the *Pool_Array* and copy them into the *Chromosome_Arrray*.
18. *If the maximum number of generations has been reached then terminate run and go to step 19. Else, go to step 11.
19. *Based on average *fv* recorded in index 0 of each chromosome select best 20 chromosomes, these chromosomes are referred to as the *Training_Chromosomes*.
    a. Set *Counter* = 0.
    b. *Counter* = *Counter* + 1.
    c. For chromosome number *Counter* within the *Training_Chromosomes* do the following:
        i. *Training_fv* = *Training_fv* + *Training_Chromosomes*[Counter][*0*]*.

            *Within the *Training_Chromosome* array we select each of the 20 chromosomes one at a time and retrieve its fitness value which is recorded in index 0.

        ii. If *Counter* < *20*, then go back to step 19 b. Else, $Training\_fv = \frac{Training\_fv}{20}$. Go to step 20.
20. *Record the *Training_Chromosomes* into Data.txt data file.

21. If the problem has not run 9 times, then go to step 10. Else, go to step 22.
22. If all three values for temporal discount schemes have not been run, then go to step 8. Else, go to step 23.
23. If all three values for temporal discount schemes have not been run, then go to step 5. Else, go to step 24.

Part 2: Testing (Simulation) Phase Pseudo Code

24. Set *Prob_Number* = 1.
25. Create a new *Price-Demand_Array_2* which contains 100 price and demand probabilities for each period. Set *Holder* = 0.
26. Read the *Training_Chromosome* data for the problem indicated by *Prob_Number* within the Data.txt data file.
27. Copy the *Training_Chromosomes* data into the *Testing_Chromosomes* for *Problem_Number* and reevaluate the 20 chromosomes using the new *Price-Demand_Array_2* created in step 22 to calculate a new *fv* for the chromosome.

    Modification of the DBCP-EA Evaluation Function for the SBCP-EA

    a. Set *Counter* = 0.
    b. *Counter* = *Counter* + 1.
    c. For chromosome number *Counter* do the following:
        i. Evaluate the *fv* 100 times using the 100 price and demand combination data recorded in the *Price-Demand_Array_2*.
        ii. Calculate the average *fv* based on the 100 *fv* created in previous step.
        iii. Record the average *fv* calculated in the previous step into index 0 of the chromosome.
        iv. *Testing_fv* = *Testing_fv* + *fv*.
        v. If *Counter* < 20, then go back to step 27 b. Else, $Testing\_fv = \frac{Testing\_fv}{20}$. Go to step 26.

Part 3: Test for Robustness Phase Pseudo Code

28. $\%\_Error = \left(\frac{(Training\_fv - Testing\_fv)}{Training\_fv} \right) \times 100.$
29. *NTR_Count* = *NTR_Count* + 1.
30. If SEA = SEA1 and *NTR_Count* < 5, then *Holder* = *Holder* + *%_Error* and go to step 25. Else, if SEA = SEA2, then *Holder* = *%_Error*.
31. Record *Holder* value into index 0 for *Prob_Number* in the *Testing_Chromosomes* array.

32. If *Prob_Number* < 81, then *Prob_Number* = *Prob_Number* + 1 and go back to step 25. Else, Done!
33. Set *Prob_Number* = 1.
34. If SEA = SEA1, then
    a. Set *Holder* equal to the value recorded in index 0 for *Prob_Number* in the *Testing_Chromosomes* array.
    b. *Holder* = *Holder*/5 (This gives *MOM_%_Error*).
    c. Record *Holder* value into index 0 for *Prob_Number* in the *Testing_Chromosomes* array.
    d. Record *Holder* value (i.e., *MOM_%_Error*) for *Prob_Number* into Table 5-6.
    e. If *Prob_Number* < 81, then *Prob_Number* = *Prob_Number* + 1 and go back to step 34 a. Else, *Prob_Number* = 1.
35. If SEA = SEA2, then
    a. Record *Holder* value (i.e., *%_Error*) for *Prob_Number* into Table 5-6.
    b. If *Prob_Number* < 81, then *Prob_Number* = *Prob_Number* + 1 and go back to step 35 a. Else, *Prob_Number* = 1.
36. Done!

5.4 SBCP-EA Results

SEA1 problem solutions were generated using an *IN* = 100. To test whether a larger sample size would result in more robust solutions, SEA2 was run using an *IN* = 500. Since the SEA1 uses a smaller sample size, the computational time was faster than the computational time for SEA2. For that reason SEA1 was tested 5 times, while SEA2 was tested only once. The *MOM_%_Error* for SEA1 is compared to the *%_Error* for SEA2. Table 5-5 shows that, in the aggregate, there is very little difference in robustness between SEA1 and SEA2 problem solutions.

A detailed breakdown of the aggregate data in Table 5-5 is shown in Table 5-6. This table shows that 49.4% (40/81) of the time SEA1 gives a smaller error and 50.6% (41/81) of the time SEA2 gives a smaller error. SEA1 gave a smaller error 55.6% (15/27) of the time for *N* = Small problems, 40.7% (11/27) of the time for *N* = Medium problems,

and 51.9% (14/27) of the time for N = Large problems. Therefore, with respect to planning horizon size (N) there doesn't seem to be a great improvement in solution robustness when SEA2 is used.

With respect to the other three parameters, SEA1 gave smaller errors 59.3% (16/27), 55.6% (15/27), and 33.3% (9/27) of the time for s = Small, Medium, and Large problems, respectively. Therefore, SEA2 does seem to give slightly more robust solutions than SEA1 when temporal discounts are large. With respect to change in price (Δp), SEA1 gave smaller errors 44.4% (12/27), 59.3% (16/27), and 44.4% (12/27) of the time for Δp = Small, Medium, and Large problems, respectively. With respect to changes in demand (Δd), SEA1 gave smaller errors 48.1% (13/27), 51.9% (14/27), and 48.1% (13/27) of the time for Δd = Small, Medium, and Large problems, respectively. For Δp and Δd there is not discernable improvement in problem solution robustness when SEA2 is run. The vast majority of the absolute difference between the errors for SEA1 and SEA2 are well below 10%. The majority of the large (around 20%) absolute difference between the errors occurs when N = Large. Overall, there is no apparent advantage associated with using SEA2 over SEA1.

Table 5-5: Comparison of Single Parameter Percentage Errors

N	***MOM_%_Error***	***%_Error***	**Difference**
Small	-20.66	-20.38	-0.28
Medium	-47.60	-45.88	-1.72
Large	-79.90	-79.16	-0.74
s	***MOM_%_Error***	***%_Error***	**Difference**
Small	-47.38	-47.50	0.12
Medium	-51.46	-51.38	-0.08
Large	-49.32	-46.55	-2.77
Δp	***MOM_%_Error***	***%_Error***	**Difference**
Small	-49.35	-47.84	-1.51
Medium	-47.57	-47.66	0.09
Large	-51.24	-49.92	-1.32
Δd	***MOM_%_Error***	***%_Error***	**Difference**
Small	-48.02	-47.96	-0.06
Medium	-50.38	-47.84	-2.54
Large	-49.76	-49.61	-0.15

Table 5-6: *MOM_%_Error* for SEA1 and *%_Error* for SEA2 Problem Solutions

s	*Δp*	*Δd*		*N*			*s*	*Δp*	*Δd*		*N*			*s*	*Δp*	*Δd*		*N*		
				Small	Medium	Large					Small	Medium	Large					Small	Medium	Large
Small	Small	Sm	SEA1	-16.00	-42.84	-69.11	Medium	Small	Sm	SEA1	-19.60	-54.95	-79.31	Large	Small	Sm	SEA1	-24.62	-43.95	-70.48
			SEA2	-17.34	-39.41	-73.94				SEA2	-20.04	-47.95	-86.68				SEA2	-21.04	-43.37	-70.32
		Md	SEA1	-24.77	-44.67	-74.75			Md	SEA1	-17.46	-46.54	-89.59			Md	SEA1	-24.55	-49.86	-83.70
			SEA2	-17.06	-39.53	-76.33				SEA2	-22.15	-48.65	-75.82				SEA2	-20.32	-38.25	-81.81
		Lg	SEA1	-20.62	-49.50	-75.94			Lg	SEA1	-18.31	-61.85	-90.59			Lg	SEA1	-19.25	-41.15	-78.38
			SEA2	-17.11	-46.03	-76.45				SEA2	-21.17	-47.12	-86.88				SEA2	-23.12	-50.66	-83.06
	Medium	Sm	SEA1	-19.67	-43.20	-79.09		Medium	Sm	SEA1	-19.93	-44.41	-91.26		Medium	Sm	SEA1	-24.49	-46.51	-67.04
			SEA2	-17.24	-43.68	-84.47				SEA2	-20.99	-50.19	-82.75				SEA2	-21.34	-44.04	-63.35
		Md	SEA1	-18.38	-41.78	-75.03			Md	SEA1	-18.62	-45.44	-77.53			Md	SEA1	-19.46	-45.03	-71.04
			SEA2	-20.66	-48.08	-75.85				SEA2	-22.70	-43.34	-81.20				SEA2	-20.83	-45.55	-55.69
		Lg	SEA1	-15.55	-43.95	-79.76			Lg	SEA1	-22.40	-42.61	-83.21			Lg	SEA1	-21.00	-43.54	-84.51
			SEA2	-17.59	-45.59	-79.20				SEA2	-20.27	-49.38	-92.01				SEA2	-22.24	-42.20	-76.48
	Large	Sm	SEA1	-18.46	-49.47	-78.78		Large	Sm	SEA1	-20.51	-57.43	-77.54		Large	Sm	SEA1	-18.54	-60.21	-59.13
			SEA2	-18.31	-42.80	-81.30				SEA2	-21.46	-49.90	-84.79				SEA2	-23.28	-44.27	-80.79
		Md	SEA1	-25.58	-33.15	-91.55			Md	SEA1	-22.60	-48.55	-100.60			Md	SEA1	-21.28	-54.51	-94.29
			SEA2	-19.37	-44.91	-94.50				SEA2	-17.92	-54.30	-78.58				SEA2	-22.33	-46.87	-79.21
		Lg	SEA1	-16.56	-47.07	-83.90			Lg	SEA1	-23.52	-49.84	-65.23			Lg	SEA1	-26.03	-53.20	-85.98
			SEA2	-18.08	-48.07	-79.58				SEA2	-21.79	-46.14	-92.99				SEA2	-24.64	-48.47	-63.26

Table 5-7 shows that SEA2 has a much longer computational time than SEA1, especially as the planning horizon increases.

Table 5-7: Average Training Computational Time (minutes)

N	**SEA1 (*IN* = 100)**	**SEA2 (*IN* = 500)**	**Difference SEA2-SEA1**
Small (per problems)	1	5	4
Medium (per problems)	7	17	10
Large (per problems)	20	85	65

From the data in Tables 5-5, 5-6 and 5-7 it was determined that it was not necessary to run the SEA2 more than once. Since the SEA2 error terms were close to those of SEA1, and the average SEA2 computational time is 3.8 times longer on average than that for SEA1, there is no advantage to training and testing problem solutions when *IN* is increased from 100 to 500. Since there is only a slight difference between the SEA1 and SEA2 problem solutions, only SEA1 results will be discussed in the remainder of this section. Future research, perhaps using a faster algorithm, could examine the impact of using an even larger *IN* (e.g., greater than 1,000) in an effort to improve problem solution robustness.

Table 5-8 shows *MOM_%_Error* for all 81 problem solutions from SEA1. Clearly as *N* increases from Small to Large, the *MOM_%_Error* also increases from an average of 20.66% for *N* = Small, to 47.2% for *N* = Medium, and 79.2% for *N* = Large. This increase in the error is expected since as the planning horizon gets longer the number of possible contracts increases very quickly, thereby greatly increasing the search space and making solutions more sensitive to fluctuations in price and demand. For this reason, practitioners should not attempt to produce solutions for long period horizons, especially in a stochastic environment.

Table 5-8: *MOM_%_Error* for SEA1

s	***Probability of Δp Range***	***Probability of Δd Range***	***N***		
			Small	Medium	Large
Small	Small	Small	-16.00	-42.84	-69.11
		Medium	-24.77	-44.67	-74.75
		Large	-20.62	-49.50	-75.94
	Medium	Small	-19.67	-43.20	-79.09
		Medium	-18.38	-41.78	-75.03
		Large	-15.55	-43.95	-79.76
	Large	Small	-18.46	-49.47	-78.78
		Medium	-25.58	-33.15	-91.55
		Large	-16.56	-47.07	-83.90
Medium	Small	Small	-19.60	-54.95	-79.31
		Medium	-17.46	-46.54	-89.59
		Large	-18.31	-61.85	-90.59
	Medium	Small	-19.93	-44.41	-91.26
		Medium	-18.62	-45.44	-77.53
		Large	-22.40	-42.61	-83.21
	Large	Small	-20.51	-57.43	-77.54
		Medium	-22.60	-48.55	-100.60
		Large	-23.52	-49.84	-65.23
Large	Small	Small	-24.62	-43.95	-70.48
		Medium	-24.55	-49.86	-83.70
		Large	-19.25	-41.15	-78.38
	Medium	Small	-24.49	-46.51	-67.04
		Medium	-19.46	-45.03	-71.04
		Large	-21.00	-43.54	-84.51
	Large	Small	-18.54	-60.21	-59.13
		Medium	-21.28	-54.51	-94.29
		Large	-26.03	-53.20	-85.98

Given that the effect of *N* is very large, it is not possible to see any effects from *s*, *Δp*, or *Δd* without controlling for the effect of *N*. These three parameters, with respect to *N* = Small, Medium, and Large, will be analyzed separately.

When *N* = Small, only *s* has any effect on *MOM_%_Error*. Figure 5-2 shows that when *N* = Small and *s* = Large the *MOM_%_Error* increases. This is due to the fact that when *s* = Large the large discounts cause the SEA to select the most common contracts

which results in a commitment to longer period contracts. This strategy is advantageous when price is relatively stable. However when price is decreasing rapidly the cost per Mbps afforded from the temporal discounts associated with longer contracts may be higher than the cost per Mbps when price is decreased later in the planning horizon. Therefore, since *s* = Large problems encourage the purchase of the most common contracts these problems are most sensitive to stochastic fluctuations in price and demand. When all demand is being satisfied by a few common contracts, any stochastic fluctuations in demand could result in deficits in each period. This may force the SEA to buy expensive single period contracts, i.e., spot contracts from the market (Cheliotis, G. 2000). These single period contracts are also sensitive to stochastic price fluctuations, which makes these solutions less robust.

It was expected that *MOM_%_Errors* would increase as Δp and Δd increased from Small to Large. However there was no indication that this was true. The search space for *N* = Small is smaller than for *N* = Medium, and much smaller than for *N* = Large. Therefore Δp and Δd do not effect the quality of the solution as expected when *N* = Small.

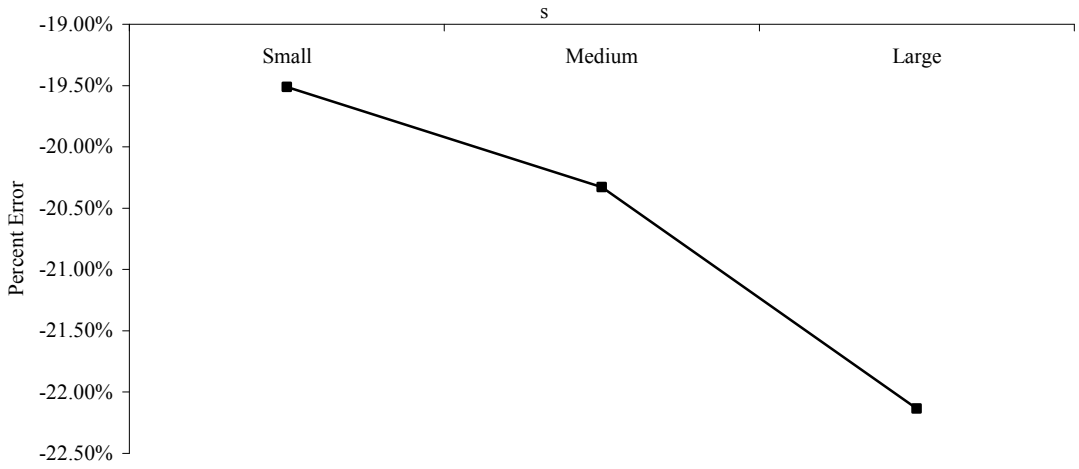


Figure 5-2: Effect of *s* on *MOM_%_Error* where *N* = Small

When *N* = Medium*, s, Δp,* and *Δd* all have an effect on *MOM_%_Error*. Figure 5-3 shows the impact of *s* on *MOM_%_Error* when *N* = Medium. For *N* = Medium, when *s* = Small the solutions were much more robust than when *s* = Medium or Large. Solutions for where *s* = Medium and *s* = Large are very similar with respect to robustness. Interestingly, this is different from what is shown in Figure 5-2 where *N* = Small. In that figure *s* = Large is very different from *s* = Small and Medium. This is because as the *N* increases from Small to Medium the search space gets larger, and the overall error terms also become larger. As the search space increases from Small to Medium, both *s* = Medium and Large problem solutions are less robust than *s* = Small problem solutions, whereas *s* = Large problem solutions have the most variability for *N* = Small problems.

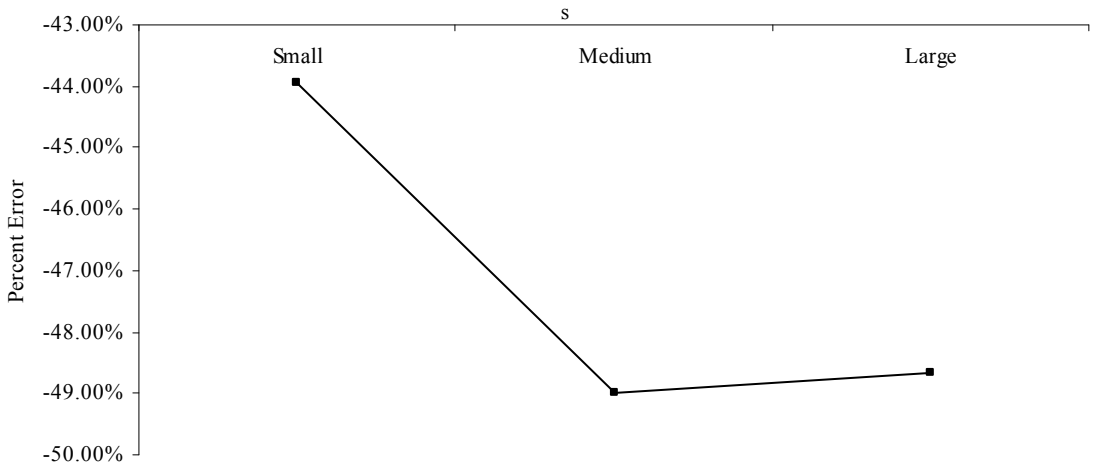


Figure 5-3: Effect of *s* on *MOM_%_Error* where *N* = Medium

Figure 5-4 shows the impact of *Δp* and *Δd* on *MOM_%_Error* when *N* = Medium. Across all three parameter settings for *Δd* the *MOM_%_Errors* when *Δp* = Small and Medium are very similar. However, the solutions become noticeably less robust when *Δd* = Small and *Δp* = Large. Common contracts produce greater penalty as *Δp* gets larger. However when *Δd* increases, single period contracts are needed to cover deficits, this reduces the penalty produced by large price fluctuations.

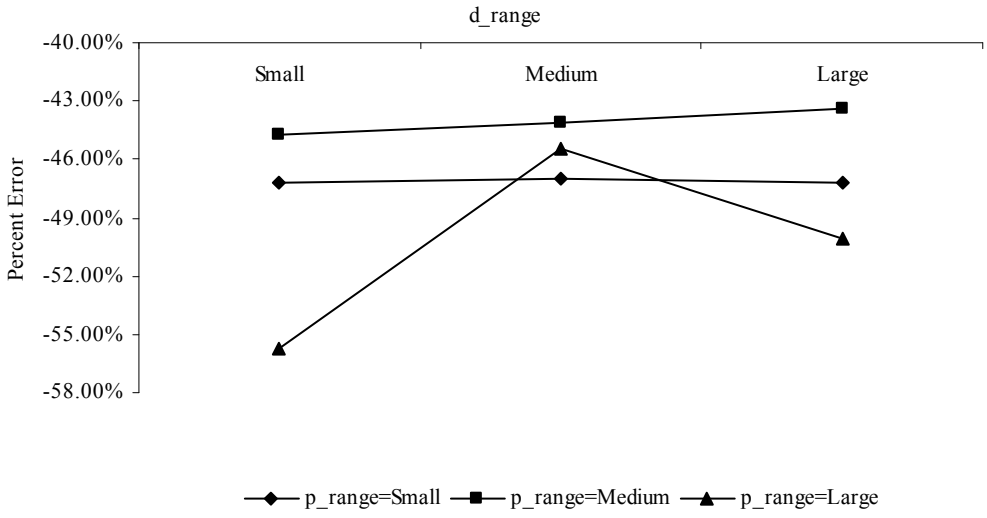


Figure 5-4: Effect of *Δp* and *Δd* on *MOM_%_Error* where *N* = Medium

Figure 5-5 shows the impact of *s* and *Δd* on *MOM_%_Error* when *N* = Large. When *N* = Large, *s* and *Δd* have an interaction, and *Δp* has no effect on *MOM_%_Error*. When *Δd* = Small and *s* = Large the *MOM_%_Errors* are much smaller than for the other parameter settings of *Δd* and *s*. This is because when *s* = Large the solutions have more common contracts and with small fluctuations in demand any deficits are also small. Therefore the resulting cost from any necessary single period contracts purchased is negligible, resulting in more robust problem solutions.

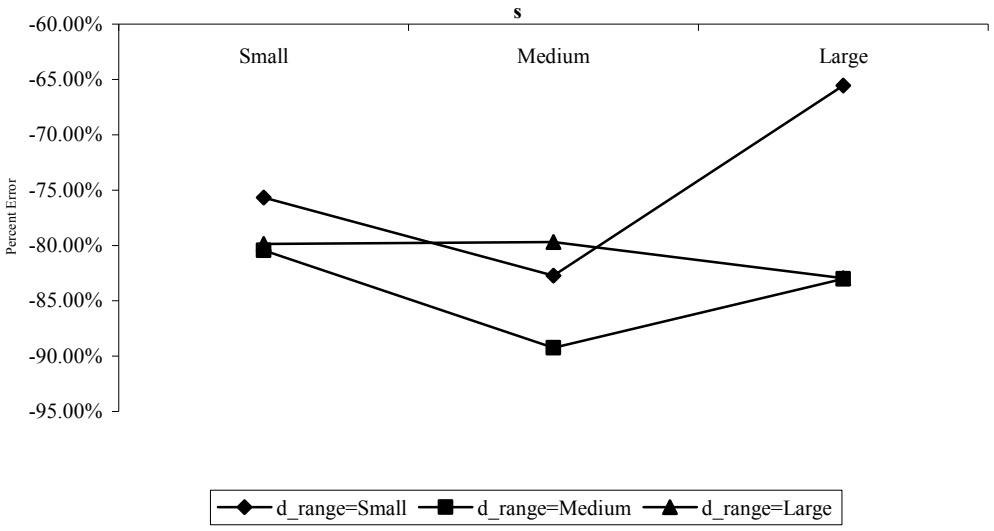


Figure 5-5: Effect of *s* and *Δd* on *MOM_%_Error* where *N* = Large

5.5 Conclusions

This chapter developed and tested an algorithm for solving the BCP in a stochastic environment. While it is not possible to compare the result from the SEA to an optimal solution, we have defined a measure of robustness for a problem solution. Robustness is measured by comparing the solutions from running the SEA on one set of price-demand instances (training) and then re-evaluating the problem solutions with another set of price-demand instances (testing) drawn from the same population. A robust solution should show little variation between the training and testing fitness values. The error measure which acts as a surrogate for robustness is *MOM_%_Error*, which is the average percentage difference between the training and testing fitness values.

The effect on problem solution robustness was also tested by increasing the number of price-demand instances from 100 to 500. There was no discernable improvement in problem solution robustness.

The results presented in this chapter show that as the planning horizon gets longer problem solution robustness decreases. This is due to the fact that the search space gets very larger quickly as the planning horizon increases. It was shown that there is some effect on the robustness of some problem solutions related to temporal discounts (s), and changes in price (Δp) and demand (Δd).

Future research could explore the impact of other demand distributions on the robustness of problem solutions. A more efficient EA could also aid in exploring the impact of wider ranges for price and demand distributions. Increasing the number of price-demand instances to a much larger number (e.g., 1000) might improve the robustness of problem solutions as well. However, with the advent of improved computational power or the development of a faster evolutionary algorithm this option might also be an avenue for future research.

CHAPTER 6 CONCLUSIONS, COMMENTS, INSIGHTS, AND FUTURE RESEARCH

In today's highly competitive environment companies need to be effective and efficient. Virtually all companies use electronic data and require a medium by which to transfer that data from one location to another. In order to accomplish this task companies need to purchase bandwidth, a medium for electronic transmission of data, provided by a third party. The cost of a bandwidth contract is dependent on the amount of bandwidth purchased (size) and the duration of the contract (length). In addition, there are often temporal discounts where the cost per unit gets considerably smaller as the duration of a contract increases.

Bandwidth purchasing is an area where companies can reduce costs of a required service. Even for small problems sizes, say a planning horizon of 6 months, the search space for possible solutions (purchasing strategies) represents a complex combinatorial problem. As the planning horizon length increases the solutions search space increases very fast. This size increase is shown in Table 2-2, where, for example, the number of contracts is 21, 78, and 171 for a 6 month, 12 month, and 18 month planning horizon, respectively.

6.1 Summary

This dissertation solves a generalized version of the problem presented by Khouja and Kumar (2004), developing and implementing an EA (a population based heuristic search

algorithm) as a means of solving the BCP. The BCP-EA allows for changes in price and demand over time.

Chapter 1 defines bandwidth, the concept of bandwidth as a commodity, and how bandwidth contract costs are assessed with respect to the purchased amount, and contract duration. Chapter 2 discusses the assumptions, constraints, and the derivation of the mathematical model that represents the BCP. Chapter 2 also includes a discussion of the data structure (i.e., the computerized representation of a problem solution) used to represent the BCP within the EA. This data structure was used for both the DEA and SEA and is shown in Table 2-1. The remainder of the chapter discusses the necessary elements that make up an EA.

Chapter 3 began with a detailed description of the important characteristics of the BCP-EA. These important characteristics include: (1) The genetic representation of a problem which translates into the computerized representation of a chromosome, or solution. (2) The method used to create the initial population of solutions. The quality of the initial population can influence the quality of the final solutions, as well as the time required to find the final solutions. Knowledge about the problem is usually considered when devising the procedure for creating the initial population. (3) The genetic operators, which are procedures that perturb existing solutions in order to create new solutions, or offspring. (4) The evaluation function, which provides a means of assessing the relative "goodness" of a given solution. (5) The selection method specifies how solutions will be selected either for the purpose of propagation (i.e., creating new solutions), or as a means of transferring selected solutions from one generation to the next. (6) The generational policy, which specifies how to select a subset of solutions

from a larger group of available solutions, where the subset group will be used as the population for the next iteration. (7) The terminating conditions, which specifies the conditions that, when met, will cause the EA to stop. (8) The EA parameters refers to the setting used within the EA such as the population size, the maximum number of generations, the probability of mutation or crossover, etc. Parameter settings can be static or dynamic. (9) Constraint handling techniques provide a procedure which transforming infeasible solutions into feasible solutions.

Chapter 4 outlines the specifications for the DBCP-EA. Figure 4-1, which shows the DBCP-EA flowchart, and the DBCP-EA pseudo code give a detailed description of the processes within the EA, as well as the order in which these processes are completed. Each of the unique elements of the DBCP-EA are discussed in detail. The experimental design for testing the DBCP-EA is illustrated in Figure 4-9. The MILP mathematical formulation is shown in Figure 4-10 and an example of an MILP output is shown in Table 4-4. The DBCP-EA and the MILP were run for the 81 problems identified in the experimental design. The best solution created by the DBCP-EA for each problem is then compared to the optimal solution found by the MILP. Table 4-5 shows the results of three example problems. For the first two example problems, with planning horizons of 6 months and 12 months, respectively, DBCP-EA performed very well when compared to the MILP. For the example problem with an 18 months horizon the DBCP-EA did not perform as well, with an error slightly less than 0.25%.

The computational time differential between the DBCP-EA and the MILP for the 6 month planning horizon was negligible, for the 12 month planning horizon the DBCP-EA took longer to find a solution than the MILP, yet the time is only slightly more than 1

minute. At 18 month the DBCP-EA is in many instances competitive with the MILP with respect to computational time, and sometimes better, as shown in Table 4-5. This fact illuminates one advantage of an EA over the MILP. That is, the DBCP-EA performs reasonably well for most problems, and it is expected that the DBCP-EA would continue to perform well with respect to solution quality and computational time for planning horizons longer than those tested. In fact, as the planning horizon gets longer the DBCP-EA may provide good solutions much faster than the MILP.

For 68 out of the 81 problems (84%) the error was less than 1%. The other thirteen problems had an average percentage error greater than 1%. Table 4-12 shows the DBCP-EA errors for the 5 runs for each of the thirteen problems. For many of the thirteen problems, there were no assignable common reasons for the occurrence of large percent errors (i.e., greater than 1%). However, when comparing the DBCP-EA results for the thirteen problems with the optimal solutions identified by the MILP, it was clear that the propensity of the repair function to assign deficit bandwidth amounts to the most common contracts (i.e., the longest duration contract available for purchase in a period) to transform infeasible solutions into feasible solutions causes suboptimal solutions.

In an effort to expand our knowledge of the effect of the parameters (N, s, p, and d) on the quality of the DBCP-EA solutions, results were examined in the aggregate. The planning horizon parameter is the greatest determinant of solution quality. As the planning horizon increases the solution quality deteriorates. This is undoubtedly due to the fact that as the planning horizon increases the solutions search space becomes dramatically larger. Since the DBCP-EA is essentially a directed random search of a

solution's search space, as the search space becomes increasingly larger, it becomes more difficult for the DBCP-EA to stumble upon an optimal solution.

It might be intuitive to think that the best strategy is to purchase the longest available contracts during any period, thereby acquiring the lowest per unit price, which will affect as many subsequent periods as possible. In some situations this is a viable option, however this dissertation has shown that in many cases in order to arrive at the lowest cost purchasing strategy purchasing the longest available contract will not provide an optimal solution. This is especially noticeable when price is declining rapidly over time. In this case, the cost savings associated with temporal discounts (i.e., a reduced per unit cost when longer duration contracts are purchased) may turn out to be more costly than if the necessary bandwidth had been purchased as late as possible in the planning horizon when the price of bandwidth has been greatly reduced. The added cost associated with bandwidth surpluses (i.e., unnecessary, wasted bandwidth capacity) and bandwidth deficits (i.e., shortages in required bandwidth) also need to be factored into the purchasing decision.

While surpluses incur added cost, in some cases a surplus in early periods can be part of the lowest purchasing strategy. This is because by purchasing more bandwidth than is required in a given period(s) the cost per unit due to temporal discounts could be less than the per unit cost applied to smaller bandwidth size contracts purchased later in the planning horizon, even when price is decreasing with time. An example of this phenomenon is illustrated in Table 4-28 where the optimal solution contains a surplus in the first period. On the other hand, deficits are strictly not allowed. When a deficit arises, the repair function either adds the amount to the most common contract (possibly

creating surpluses in other time periods) or the least common contract (the shortest commitment, yet most expensive option). The cheaper of the two options is chosen. This repair function works well with most problems (68 out of 81) but in a few problems, it produced solutions with a 1% or greater error. An increase in percentage error is especially likely to occur when the optimal solution stipulates the purchase of mid-range contracts (i.e., contracts that are neither the most common nor the least common in a period).

In Chapter 5 the EA is modified to incorporate stochastic changes in the price and demand. A uniform distribution with three difference ranges was used to generate instances of price and demand. Purchasing strategies in a stochastic environment needs to be resilient, providing good solutions regardless of stochastic changes in price and demand. Therefore, a good problem solution, i.e., robust, would consistently perform well over different distribution ranges for price and demand.

For each problem, a probability distribution range is chosen for price and demand, along with other parameter settings. The price and demand probability distribution ranges used by the SEA is given in Table 5-1. The SEA uses a two-dimensional array, *Price-Demand_Array_1*, of width 200 (100 price instances and 100 demand instances) and of a length equal to the planning horizon for that problem. The SEA is run in the training phase of the SBCP-EA. At the end of the training phase the 20 best solutions for each problem are recorded and the 20 fitness values are averaged together to obtain an overall solution fitness value, referred to as the *Training_fv*. Following the training phase a new two-dimensional array, *Price-Demand_Array_2*, is populated with a new set of 100 price-demand instances. The 20 best solutions for each problem are then re-

evaluated using the *Price-Demand_Array_2* during the testing phase. At the end of the testing phase the fitness values of 20 best solutions for each problem are averaged together to obtain an overall problem solution fitness value, referred to as the *Testing_fv*. The average percent difference between the *Training_fv* and the *Testing_fv* for each problem represents a solution's measure of robustness. This is called the testing for robustness phase of the SEA. A detailed discussion of the three phases of the SEA can be found in Section 5.2. Figure 5.1 shows the flowchart of the SEA process, and the SEA pseudo code is presented in Section 5.3.

Within the SBCP-EA the SEA was run twice. The first time, the SEA was run using 100 price-demand instances (SEA1). The resulting 20 best solutions for each problem were re-evaluated in the testing phase five separate times, where each time a different *Price-Demand_Array_2* was used. The robustness for each SEA1 problem solutions showed little variation between the five runs. Table 5-8 shows that as the planning horizon gets longer problem solution's become less robust; that is to say, that the percentage difference becomes larger. These are the same results found in the DBCP-EA section where the solution quality for each problem also decreased as the planning horizon increased.

SEA (training phase) was run a second time with 500 instances (SEA2) to see if a larger number of price-demand instances would result in greater problem solution robustness. The robustness for each of the problem solutions was then calculated. The results in Table 5-6 show that there was no added benefit to running the SEA for greater number of price-demand instances.

The length of the planning horizon is the only parameter that consistently and noticeably affects the quality of the results. Shorter planning horizon solutions are much more robust than solutions for larger planning horizons. For that reason, it is advisable that practitioners not attempt make financial commitments based on the purchasing strategies obtained using long planning horizon problem formulations. Future research in this area may provide the tools and techniques that can improve upon the results obtained in this dissertation, thereby providing more robust and accurate low cost purchasing strategies.

6.2 Contributions

The five main contributions from this work are enumerated below.

1. Extended the original model by Khouja and Kumar (2004) by allowing overlapping contracts and changes in price and demand over time.
2. Formulated an MILP to find optimum solutions for the BCP problem.
3. Designed and implemented an EA to find good solutions for the BCP in deterministic and stochastic environments where there is no restriction on the behavior of price, demand, or temporal discounts.
4. Demonstrated that the DEA performed well in a deterministic environment producing optimal, or near optimal, solutions in most cases, especially for short and medium planning horizons.
5. Modified the DEA to a stochastic environment and performed an exploratory study to examine the robustness of problem solutions.

The EA is able to solve very large problems, whereas CPLEX was unable to identify the optimal solutions in a reasonable amount of time (i.e., days) for planning

horizons of 24 months. These runs were terminated due to the excessive time and computation resources that were required. We have also shown that the BCP-EA can be used in a stochastic environment, which allows for the modeling of change in prices and demand that is not mathematically tractable.

6.3 Future Research

As mentioned earlier, the repair function employed by the BCP-EA has been shown to unintentionally bias infeasible solutions as they are transformed into feasible solutions. Often the repair function adds deficit amounts for a period to the contract that has the greatest effect of increasing bandwidth for subsequent periods in the planning horizon (i.e., most common contracts). It is believed that since the repair function is biased and a substantial amount of chromosomes undergo repair, the procedures used in the repair function restrict and limit the exploration of the problem's search space.

One pattern of good solutions that was seen in the optimal MILP solutions is that periods at the beginning of the planning horizon often purchase mid-range contracts (i.e., contracts that are neither most common nor least common, but somewhere in between). For those periods that fall within the middle and end of the planning horizon the optimal strategy is often to purchase contracts with the longest durations possible (i.e., most common contracts). This knowledge may be useful for future research when a new repair function will be developed that will not negatively affect solution quality. If a new, improved repair function can not be created, then it might be more efficient to simply discard infeasible solutions. The ultimate goal is to reduce the overall percentage error for the problem solutions obtained using the DBCP-EA and the SBCP-EA by

eliminating, or reducing, the apparent negative bias imposed by the existing repair function.

Even though the BCP-EA was not designed with speed as a required criterion, it was found that an added benefit of the BCP-EA was that in some cases it identified a "good" solution faster than the MILP for 18 period planning horizon problems. Therefore, the BCP-EA is very attractive when solving large problems. It is believed that the existing BCP-EA could be streamlined and optimized to reduce its computational time further so that "good" solutions can be identified in much less time.

Along with making the BCP-EA faster, it might also be advantageous to examine the impact of a larger number of price-demand instances has on the measure of robustness in a stochastic environment thereby potentially increasing solution robustness. However, the number of price-demand instance that can practically be used will be determined by available computational speed, data storage resources, and time constraints.

In an extension of this dissertation, future research could focus on redesigning the SBCP-EA to examine the effect of other probability distributions; for example, normal and exponential. Another area of future research that might provide more robust solutions for the SBCP-EA would be to alter the way that the training phase is performed. Presently, the same 100 price-demand instance array is used to evolve and evaluate the 20 best problem solutions across all generations of the run of a problem. Future research might incorporate a more complex procedure. For example, for each generation a new set of 100 price-demand instances could be used to evolve and evaluate the chromosomes. This procedure has the advantage of introducing more diversity and

randomness in identifying ‘good’ solutions. Another possibility would be to use a unique 100 price-demand instance for each chromosome within each generation. This procedure has the advantage of being truly random, yet would be very computationally intensive.

REFERENCES

ADSL, http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci213764,00.html, 04/04/04.

Azadivar, F., and Wang, J. J. 2000. Facility layout optimizaton using simulation and genetic algorithms. International Journal of Production Research, 38(17): 4369-4383.

Balakrishnan, A., Magnanti, T. L., and Mirchandani, P. 1998. Designing hierarchical survivable networks. Operations Research, 46(116-136).

Balakrishnan, A., Magnanti, T. L., and Wong, R. T. 1995. A dual-ascent procedure for large scale uncapacitated network design. Operations Research, 37: 716-740.

Bandwidth, http://compnetworking.about.com/library/glossary/bldef-bandwidth.htm?terms=bandwidth, 04/04/04.

Bandwidth Finders, http://www.bandwidthfinders.com/, 04/04/04.

Band-X, http://www.band-x.com/en/, 04/04/04.

Borthick, S. 2001. Cheap bandwidth: How low can it go? Business Communications Review, 31(8): 14-16.

Brown, K., and Norgaard, R. 1992. Modeling the telecommunications pricing decision. Decision Sciences, 23(3): 673-687.

Burnetas, A., and Gilbert, S. 2001. Future capacity procurements under unknown demand and increasing cost. Management Science, 47(7): 979-992.

Chang, S., and Gavish, B. 1995. Lower bounding procedures for multi-period telecommunications network expansion problems. Operations Research, 43(1): 43-57.

Cheliotis, G. 2000. Bandwidth Trading in the Real World: Findings and Implications for Commodities Brokerage. Reseach Report (+93290)Computer Science/Mathematics, IBM Research, Zurich Research Laboratory, 8803 Ruschlikon, Switzerland.

Chen, B., Hassin, R., and Tzur, M. 2002. Allocation of bandwidth and storage. IIE Transactions, 34(5): 501-507.

Cochi, R., Shenkar, S., Estrin, D., and Zhang, L. 1993. Pricing in computer networks: Motivation, formulation and example. IEEE/ACM Transactions on Networking, 1: 614-627.

Computer King, www.thecomputerking.com, 04/04/04.

CPLEX (1995). *Using the CPLEX Callable Library.* Incline Village, NV: CPLEX Optimization, Inc.

Dense Wavelength Division Multiplexing, http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci213892,00.html, 04/04/04.

Econmides, N. 1996. The economics of networks. International Journal of Industrial Organization, 16: 675-699.

Fiber Optic, http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci212113,00.html, 04/04/04.

FitzGerald, J., and Dennis, A. 1999. Business Data Communications and Networking. New York, NY: John Wiley & Sons, Inc.

Galbi, D. A. 2001. Growth in the "new economy": US bandwidth use and pricing across the 1990s. Telecommunications Policy, 25: 139-154.

Gavish, B. 1992. Topological design of computer communication networks--the overall design problem. European Journal of Operational Research, 58: 149-172.

Horak, R. 2000. Communications Systems and Networks (2nd ed.). New York, NY: M&T Books.

Keon, N. J., and Anandalingam, A. 2003. Optimal Pricing for Multiple Services in Telecommunications Networks Offering Quality of Service Guarantees. IEEE Transactions on Networking, 11: 66-80.

Khouja, M. J., and Kumar, R. L. 2004. Acquisition of Telecommunications Bandwidth Under Economies of Scale in Size and Duration of Contracts. Decision Sciences, 36(1): 135-158.

Laudon, K. C., and Laudon, J. P. 2003. Essentials of Management Information Systems: Managing the Digital Firm (5th ed.). New Jersey: Prentice Hall.

Michalewicz, Z. 1996. Genetic algorithms + data structures = evolution programs (3rd ed.). New York, NY: Springer-Verlag.

Paris, J.-L., and Pierreval, H. 2001. A distributed evolutionary simulation optimization approach for the configuration of multiproduct kanban systems. International Journal of Computer Integrated Manufacturing, 14(5): 421-430.

Pierreval, H., and Tautou, L. 1997. Using evolutionary algorithms and simulation for the optimization of manufacturing systems. IIE Transactions, 29(3): 181-189.

Premkumar, G., and Chu, C. H. 2000. Telecommunications network design--comparison of alternative approaches. Decision Sciences, 31(2): 483-507.

Reeves, C. R. 1997. Genetic algorithms for the operations researcher. INFORMS Journal on Computing, 9: 231-250.

Satellite, http://searchmobilecomputing.techtarget.com/sDefinition/0,,sid40_gci212939,00.html, 04/04/04.

Shapiro, C., and Varian, C. 1999. Information Rules: A strategic guide to the network economy. Boston, MA: Harvard Business School Press.

SPSS 13.0 for Windows, 2004. SPSS Inc., Chicago, Illinois 60606.

Synchronous Optical Network (SONET) Transmission, http://www.iec.org/online/tutorials/sonet_trans/topic05.html, 04/04/04.

T1, http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci213084,00.html, 04/04/04.

TeleGeography, http://www.telegeography.com/, 04/04/04.

Vose M. D., and Liepins, G. E. 1991. Punctuated equilibria in genetic search. Complex Systems, 5: 31 44.

Williams, H. Paul 2005. Model Building in Mathematical Programming (4th ed.). New York, NY: John Wiley & Sons Ltd., pages 144-153 (Chapter 8).

APPENDIX A: SOURCE CODE FOR DBCP-EA

```
package ea_improved;
/** imports**/
import java.util.*;
import java.lang.*;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.*;
import java.lang.Math.*;

public class ea
{
 /** initialize variables **/
 int NumPeriods;
 int PopSize;
 int maxGenerations;
 double OnePointCrossoverRate;
 double UniformCrossoverRate;
 double MutationRate1;
 double MutationRate2;
 int gen;
 double[][] Chromosome;
 Random Rand_Number;
 ReadFile read;
 WriteFile write;
 double NumContracts;
 int n_scale;
 int IntNumContracts;
 double[] B = new double[3];
 String temp;
 String temp2;
 StringBuffer buff;
 StringBuffer buff2;
 double rp;
 double rd;
 double gp;
 double ad;
 double gd;
 double B1;
 double B2;
 double B3;
 double iLP_Opt;
 double[] rp_Array;
 double[] rd_Array;
```

```
double[] D;
int[] Contract_Length;
double[] TCQ;
double[] TCX;
int[] Contract_Period;
int[] NumContracts_EachPeriod;
int feasibilityFlag = 0;
int feasibilityFlagPool = 0;
int indexHolder;
double[] bestChromosome;
double[][] Pool;
int PoolSize;
int counter;
int NumOffspring;
double Date;
int FileNum = 0;
int change;
double[] Best_All_Runs;
int Repair1;
int Repair2;
int RepairTie;
int AlreadyDone = 0;

Vector Fill_Best = new Vector();

public ea()
{
 this.Rand_Number = new Random();
 this.read = new ReadFile("input.txt");
 this.buff = new StringBuffer();
 this.buff2 = new StringBuffer();
 try
 {
  jbInit();
 }
 catch (Exception ex)
 {
  ex.printStackTrace();
 }
}

private void read_input_file()
{
 System.out.println("DATE: " + Date);
 temp = "DATE: " + Date;
 buff.append(temp + "\r\n");
```

```
  System.out.println("N = " + NumPeriods + ", gp = " + gp + ", ad = " + ad +
                     ", gd = " + gd + ", B1 = " + B1 + " , B2 = " + B2 +
                     ", B3 = " + B3);
  temp = "N = " + NumPeriods + ", gp = " + gp + ", ad = " + ad + ", gd = " +
     gd + ", B1 = " + B1 + " , B2 = " + B2 + ", B3 = " + B3;
  buff.append(temp + "\r\n");

  NumContracts = .5 * NumPeriods * (NumPeriods + 1);
  IntNumContracts = new Double(NumContracts).intValue();

  System.out.println("Number of contracts =" + IntNumContracts +
                     ", NumPeriods = " + NumPeriods);
  temp = "!Number of contracts =" + IntNumContracts;
  buff.append(temp + "\r\n");

  System.out.println("LP Optimal =" + iLP_Opt);
  temp = "LP Optimal =" + iLP_Opt;
  buff.append(temp + "\r\n");
}

private void fill_arrays()
{
  double product = 1;
  int count;
  int i;
  int j;
  int index1;
  int enumerate1;
  int enumerate2;
  Contract_Period = new int[IntNumContracts];
  Contract_Length = new int[IntNumContracts];
  Pool = new double[PoolSize][IntNumContracts + 1];
  int p = new Integer(Rounding.toString(gp * 10, 0)).intValue();
  int ad_d = new Integer(Rounding.toString(ad * 1, 0)).intValue();
  int d = new Integer(Rounding.toString(gd * 10, 0)).intValue();
  rp = 1 - gp;
  rd = 1 + gd;
  rp_Array = new double[NumPeriods];
  rd_Array = new double[NumPeriods];
  D = new double[NumPeriods];
  NumContracts_EachPeriod = new int[NumPeriods];

  rp_Array[0] = 1;
  for (count = 1; count < NumPeriods; count++)
  {
    product = product * rp;
```

```
    rp_Array[count] = product;
  }
  product = 1;
  rd_Array[0] = 1;
  for (count = 1; count < NumPeriods; count++)
  {
    product = product * rd;
    rd_Array[count] = product;
  }
  D[0] = ad;
  for (count = 1; count < NumPeriods; count++)
  {
    D[count] = new Double(Math.ceil(ad * rd_Array[count])).intValue();
  }
  for (i = 0; i < NumPeriods; i++)
  {
    NumContracts_EachPeriod[i] = (NumPeriods - (i)) * (i + 1);
  }
  index1 = 0;
  for (enumerate1 = 0; enumerate1 <= NumPeriods; enumerate1++)
  {
    for (enumerate2 = 1; enumerate2 <= NumPeriods - enumerate1;
        enumerate2++)
    {
      Contract_Length[index1] = enumerate2;
      index1 = index1 + 1;
    }
  }
  count = 0;
  for (i = 1; i < NumPeriods + 1; i++)
  {
    for (j = 1; j < NumPeriods + 2 - i; j++)
    {
      Contract_Period[count] = i;
      count = count + 1;
    }
  }
}

private void initialize_chromosomes()
{
  double prob_purchase = .5;
  double Num_Contracts = NumPeriods * .5 * (NumPeriods + 1);
  Chromosome = new double[PopSize][IntNumContracts + 1];
  int j;
  int k;
```

```
  double number;
  double tempz;
  double tempzz;
  double tempy;
  int number2;
  int x;
  int y;

  for (k = 0; k < PopSize; k++)
  {
   Chromosome[k][0] = 999999;
   for (j = 1; j <= IntNumContracts; j++)
   {
    number = Rand_Number.nextDouble();
    if (number <= prob_purchase)
    {
     tempz = Rand_Number.nextDouble() * 100;
     tempzz = Rand_Number.nextDouble() * 10;
     tempy = (tempz * tempzz) / 2;
     number2 = new Integer(Rounding.toString(tempy, 0)).intValue();
     Chromosome[k][j] = number2;
    }
    else
    {
     Chromosome[k][j] = 0;
    }
   }
  }
}

public void print_Chromosome()
{
  int start;
  int start1;
  System.out.println("Print Pool");
  for (start = 0; start < PopSize; start++)
  {
   System.out.print(Chromosome[start][0] + " | ");
   System.out.println();
  }
  System.out.println();
}

public void print_Chromosome(int indexHolder)
{
  System.out.println("Repaired Chromosome: ");
```

```
  for (int k5 = 0; k5 < this.IntNumContracts; k5++)
  {
   System.out.print(this.Pool[indexHolder][k5] + " | ");
  }
  System.out.println("\n");
}

private void initialize_chromosome_pool(int PopSize, int IntNumContracts)
{
  int i;
  int j;
  for (i = 0; i < PopSize; i++)
  {
   for (j = 0; j < IntNumContracts + 1; j++)
   {
    Pool[i][j] = Chromosome[i][j];
   }
  }
}

private void OnePoint_crossover_operator()
{
  double randNum;
  int cutPoint;
  int[] OffSpringOne = new int[IntNumContracts];
  int[] OffSpringTwo = new int[IntNumContracts];
  int[] ParentOne = new int[IntNumContracts];
  int[] ParentTwo = new int[IntNumContracts];
  int i;
  int j;
  int x;
  int z;
  int num1;
  int num2;
  Vector tabu = new Vector();
  for (int a = 0; a < PopSize; a++)
  {
   tabu.addElement(new Integer(a));
  }

  for (x = 0; x < PopSize / 2; x++)
  {
   randNum = Rand_Number.nextDouble();
   if (randNum < OnePointCrossoverRate)
   {
    randNum = Rand_Number.nextDouble();
```

```
randNum = randNum * (IntNumContracts - 1) + 1;
cutPoint = new Integer(Rounding.toString(randNum, 0)).intValue();
cutPoint = cutPoint - 1;
num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
if (Pool[num1][0] > Pool[num2][0])
{
 for (z = 1; z < IntNumContracts + 1; z++)
 {
  ParentOne[z - 1] = new Double(Pool[num2][z]).intValue();
 }
}
else
{
 for (z = 1; z < IntNumContracts + 1; z++)
 {
  ParentOne[z - 1] = new Double(Pool[num1][z]).intValue();
 }
}
num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
if (Pool[num1][0] > Pool[num2][0])
{
 for (z = 1; z < IntNumContracts + 1; z++)
 {
  ParentTwo[z - 1] = new Double(Pool[num2][z]).intValue();
 }
}
else
{
 for (z = 1; z < IntNumContracts + 1; z++)
 {
  ParentTwo[z - 1] = new Double(Pool[num1][z]).intValue();
 }
}
for (i = 0; i < cutPoint; i++)
{
 OffSpringOne[i] = ParentOne[i];
 OffSpringTwo[i] = ParentTwo[i];
}
for (i = cutPoint; i < IntNumContracts; i++)
{
```

```
        OffSpringOne[i] = ParentTwo[i];
        OffSpringTwo[i] = ParentOne[i];
      }
      //*****************
       for (i = 1; i < IntNumContracts + 1; i++)
       {
         Pool[counter][i] = OffSpringOne[i - 1];
         Pool[counter + 1][i] = OffSpringTwo[i - 1];
       }
      counter = counter + 2;
      NumOffspring = NumOffspring + 2;
    }
  }
}

private void Uniform_crossover_operator()
{
  int i;
  int j;
  int k;
  int x;
  int z;
  double randNum;
  int decisionVariable;
  int[] scrambleArray = new int[IntNumContracts];
  int[] OffSpringOne = new int[IntNumContracts];
  int[] OffSpringTwo = new int[IntNumContracts];
  int[] ParentOne = new int[IntNumContracts];
  int[] ParentTwo = new int[IntNumContracts];
  int decision;
  int num1;
  int num2;
  Vector tabu = new Vector();

  for (int a = 0; a < PopSize; a++)
  {
   tabu.addElement(new Integer(a));
  }
  for (x = 0; x < PopSize / 2; x++)
  {
    for (i = 0; i < IntNumContracts; i++)
    {
      randNum = Rand_Number.nextDouble();
      if (randNum < UniformCrossoverRate)
      {
        scrambleArray[i] = 1;
```

```
  }
  else
  {
   scrambleArray[i] = 0;
  }
}
num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
if (Pool[num1][0] > Pool[num2][0])
{
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentOne[z - 1] = new Double(Pool[num2][z]).intValue();
  }
}
else
{
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentOne[z - 1] = new Double(Pool[num1][z]).intValue();
  }
}
num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
if (Pool[num1][0] > Pool[num2][0])
{
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentTwo[z - 1] = new Double(Pool[num2][z]).intValue();
  }
}
else
{
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentTwo[z - 1] = new Double(Pool[num1][z]).intValue();
  }
}
for (k = 0; k < IntNumContracts; k++)
{
  decision = scrambleArray[k];
  if (decision == 0)
```

```
        {
          OffSpringOne[k] = ParentOne[k];
          OffSpringTwo[k] = ParentTwo[k];
        }
        else if (decision == 1)
        {
          OffSpringOne[k] = ParentTwo[k];
          OffSpringTwo[k] = ParentOne[k];
        }
      }
      //*****************

       for (i = 1; i < IntNumContracts + 1; i++)
       {
         Pool[counter][i] = OffSpringOne[i - 1];
         Pool[counter + 1][i] = OffSpringTwo[i - 1];
       }
      counter = counter + 2;
      NumOffspring = NumOffspring + 2;
    }
  }

  private void mutation_operator()
  {
//Mutation #2: Local Seach with Tabu List
    int x;
    int z;
    int i;
    int j;
    int k;
    double randNum;
    int decisionVariable;
    int[] scrambleArray = new int[IntNumContracts];
    int[] OffSpring = new int[IntNumContracts];
    int[] Parent = new int[IntNumContracts];
    int decision;
    int num1;
    int num2;
    int P1 = 999;
    double tempz = 0;
    double tempzz;
    int number2;
    int number3;
    Integer number4;
    int listSize;
    int redo_flag = 0;
```

```
Vector list = new Vector();
Vector tabu = new Vector();

for (int a = 0; a < PopSize; a++)
{
 tabu.addElement(new Integer(a));
}
for (x = 0; x < PopSize; x++)
{
 for (i = 0; i < IntNumContracts; i++)
 {
  randNum = Rand_Number.nextDouble();
  if (randNum < MutationRate2)
  {
   scrambleArray[i] = 1;
  }
  else
  {
   scrambleArray[i] = 0;
  }
 }

//_____________________________________________________________

  randNum = Rand_Number.nextInt(tabu.size());
  int temp = new Double(randNum).intValue();
  num1 = ( (Integer) tabu.elementAt(temp)).intValue();
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   Parent[z - 1] = new Double(Pool[num1][z]).intValue();
   OffSpring[z - 1] = new Double(Pool[num1][z]).intValue();
  }
  number4 = new Integer(num1);
  list.addElement(number4);
  for (k = 0; k < IntNumContracts; k++)
  {
   decision = scrambleArray[k];
   if (decision == 0)
   {
    OffSpring[k] = Parent[k];
   }
   else if (decision == 1)
   {

    int size1;
```

```
int Purchased = 0;
double cumulativeTotal = 0;
int index2;
int period = 0;
int index3;
double Total;
double tempy1;
double tempy2;
double tempy3;
int turns;
int numx;

TCQ = new double[IntNumContracts];
TCX = new double[IntNumContracts];
for (j = 0; j < IntNumContracts; j++)
{
 index2 = Contract_Period[j];
 if (j == k)
 {
  period = index2;
 }
 index3 = Contract_Length[j];
 size1 = OffSpring[j];
 TCQ[j] = rp_Array[index2 - 1] * ( (B[0] * index3) + B[2]);
 if (size1 > 0)
 {
  Purchased = 1;
 }
 else
 {
  Purchased = 0;
 }
 TCX[j] = rp_Array[index2 - 1] * B[1] * index3;

 Total = 0;
 Total = TCQ[j] * size1 + TCX[j] * Purchased;

 cumulativeTotal = cumulativeTotal + Total;
}

tempy1 = cumulativeTotal;
tempy3 = tempy1;
tempy2 = 0;
turns = 0;
while (tempy2 < tempy1)
{
```

```
cumulativeTotal = 0;
tempy1 = tempy3;
turns = turns + 1;
randNum = Rand_Number.nextDouble();
randNum = randNum *
   ( (D[period - 1] * Rand_Number.nextDouble() * .3) - 1) + 1;
numx = new Integer(Rounding.toString(randNum, 0)).intValue();
numx = numx - 1;
randNum = Rand_Number.nextDouble();
if (Parent[k] > 0)
{
 if (randNum <= .8)
 {
  tempz = Parent[k] + (numx * -1);
  if (tempz < 0)
  {
   tempz = 0;
  }
 }

 number2 = new Integer(Rounding.toString(tempz, 0)).intValue();
 OffSpring[k] = number2;
}

TCQ = new double[IntNumContracts];
TCX = new double[IntNumContracts];

for (j = 0; j < IntNumContracts; j++)
{
 index2 = Contract_Period[j];
 index3 = Contract_Length[j];
 size1 = OffSpring[j];
 TCQ[j] = rp_Array[index2 -
    1] * ( (B[0] * index3) + B[2]);
 if (size1 > 0)
 {
  Purchased = 1;
 }
 else
 {
  Purchased = 0;
 }
 TCX[j] = rp_Array[index2 - 1] * B[1] * index3;

 Total = 0;
 Total = TCQ[j] * size1 + TCX[j] * Purchased;
```

```
            cumulativeTotal = cumulativeTotal + Total;
          }
          tempy2 = cumulativeTotal;
          tempy3 = tempy2;
          if (turns == 20)
          {
            OffSpring[k] = Parent[k];
            tempy1 = 0;
          }
        }
        turns = 0;
      }
    }
    //*****************

     for (i = 1; i < IntNumContracts + 1; i++)
     {
       Pool[counter][i] = OffSpring[i - 1];
     }
    counter = counter + 1;
    NumOffspring = NumOffspring + 1;
  }
}

private void mutation_operator_original()
{
  //Mutation #1: Tabu List with decreases in randomly selected genes *Not Used
  int x;
  int z;
  int i;
  int j;
  int k;
  double randNum;
  int decisionVariable;
  int[] scrambleArray = new int[IntNumContracts];
  int[] OffSpring = new int[IntNumContracts];
  int[] Parent = new int[IntNumContracts];
  int decision;
  int num1;
  int num2;
  int P1 = 999;
  double tempz;
  double tempzz;
  double tempy;
  int number2;
  int number3;
```

```
Integer number4;
int listSize;
int redo_flag = 0;
int index2;
int period = 0;

Vector list = new Vector();
Vector tabu = new Vector();

for (int a = 0; a < PopSize; a++)
{
 tabu.addElement(new Integer(a));
}
for (x = 0; x < PopSize; x++)
{
 for (i = 0; i < IntNumContracts; i++)
 {
  randNum = Rand_Number.nextDouble();
  if (randNum < MutationRate1)
  {
   scrambleArray[i] = 1;
  }
  else
  {
   scrambleArray[i] = 0;
  }
 }
 //_____________________________________________________________________

 randNum = Rand_Number.nextInt(tabu.size());
 int temp = new Double(randNum).intValue();
 num1 = ( (Integer) tabu.elementAt(temp)).intValue();

 for (z = 1; z < IntNumContracts + 1; z++)
 {
  Parent[z - 1] = new Double(Pool[num1][z]).intValue();
  OffSpring[z - 1] = new Double(Pool[num1][z]).intValue();
 }
 number4 = new Integer(num1);
 list.addElement(number4);
 for (k = 0; k < IntNumContracts; k++)
 {
  decision = scrambleArray[k];
  if (decision == 0)
  {
   OffSpring[k] = Parent[k];
```

```
      }
      else if (decision == 1)
      {
       for (j = 0; j < IntNumContracts; j++)
       {
        index2 = Contract_Period[j];
        if (j == k)
        {
         period = index2;
        }
       }
       randNum = Rand_Number.nextDouble();
       tempz = randNum *
          ( (D[period - 1] * Rand_Number.nextDouble() * .3) - 1) + 1;
       number2 = new Integer(Rounding.toString(tempz, 0)).intValue();
       if (Parent[k] > 0)
       {
        if (Parent[k] > number2)
        {
         OffSpring[k] = Parent[k] - number2;
        }
        else
        {
         OffSpring[k] = 0;
        }
       }
      }
     }
     for (i = 1; i < IntNumContracts + 1; i++)
     {
      Pool[counter][i] = OffSpring[i - 1];
     }
     counter = counter + 1;
     NumOffspring = NumOffspring + 1;
    }
  }

  private void select_next_generation()
  {
   int i;
   int j;
   int x;
   int y;
   double randNum;
   int number;
   int num1;
```

```
  int num2;
  Vector tabu = new Vector();
  Chromosome[0] = this.bestChromosome.clone();
  for (int a = 0; a < NumOffspring + PopSize; a++)
  {
   tabu.addElement(new Integer(a));
  }
  for (i = 1; i < PopSize; i++)
  {
   randNum = Rand_Number.nextDouble();
   randNum = randNum * (NumOffspring + PopSize - 1) + 1;
   number = new Integer(Rounding.toString(randNum, 0)).intValue();
   number = number - 1;
   num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
      intValue();
   num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
      intValue();
   if (Pool[num1][0] < Pool[num2][0])
   {
    for (j = 0; j < IntNumContracts + 1; j++)
    {
     Chromosome[i][j] = Pool[num1][j];
    }
   }
   else
   {
    for (j = 0; j < IntNumContracts + 1; j++)
    {
     Chromosome[i][j] = Pool[num2][j];
    }
   }
  }
}

public void print_Chromosome_pool_with_FV()
{
  int start;
  int start1;
  System.out.println("Print Pool");
  for (start = 0; start < NumOffspring; start++)
  {
   for (start1 = 0; start1 < IntNumContracts; start1++)
   {
    System.out.print(Pool[start][start1] + " | ");
   }
   System.out.println("<-- PoolFitnessValue = " + Pool[start][0]);
```

```
  }
  System.out.println();
}

public void print_Chromosome_pool_withOUT_FV(int PopSize, int IntNumContracts)
{
  int start;
  int start1;
  System.out.println("Print Pool");
  for (start = 0; start < NumOffspring; start++)
  {
    for (start1 = 0; start1 < IntNumContracts; start1++)
    {
      System.out.print(Pool[start][start1] + " | ");
    }
    System.out.println();
  }
  System.out.println();
}

private void check_feasibility_pool(int NumPeriods)
{
  int k = 0;
  int i = 0;
  int j = 0;
  int temp1 = 0;
  int temp3 = 0;
  int offset;
  int index;
  int y;
  int[] Copy = new int[IntNumContracts];

  for (i = 1; i < IntNumContracts + 1; i++)
  {
    Copy[i - 1] = new Double(Pool[indexHolder][i]).intValue();
  }
  feasibilityFlagPool = 0;
  for (k = 1; k <= NumPeriods; k++)
  {
    offset = k - 1;
    index = offset;
    for (i = NumPeriods - k; i < NumPeriods; i++)
    {
      for (j = 1; j <= NumPeriods - k + 1; j++)
      {
        index = index + 1;
```

```
      temp1 = Copy[index - 1];
      temp3 = temp3 + temp1;
     }
     offset = offset - 1;
     index = index + offset;
    }
    if (D[k - 1] > temp3)
    {
     feasibilityFlagPool = 1;
    }
    temp3 = 0;
  }
}

private void repair_function_pool(int NumPeriods)
{
  NumContracts = .5 * NumPeriods * (NumPeriods + 1);
  IntNumContracts = new Double(NumContracts).intValue();

  int k = 0;
  int i = 0;
  int j = 0;
  int temp1 = 0;
  int temp3 = 0;
  int index = 0;
  double difference = 0;
  int sizeDifference = 0;
  double[] OffSpring1 = new double[IntNumContracts + 1];
  double[] OffSpring2 = new double[IntNumContracts + 1];
  double[][] DeficitArray = new double[NumPeriods][IntNumContracts + 1];
  int[] CommonContracts = new int[IntNumContracts + 1];
  int countPeriod3 = 0;
  int k5;
  int k6;
  int holder = 0;
  int holderB = 0;
  int maxNum;
  int ConCount;
  int maxNumIndex = 0;
  double deficitAmount;
  int turns = 0;
  int numDeficit = IntNumContracts;
  int requiredRepair = 0;
  turns = 0;
  numDeficit = NumPeriods + 1;
  temp3 = 0;
```

```
index = 0;
temp1 = 0;
difference = 0;
sizeDifference = 0;
countPeriod3 = 0;
holder = 0;
ConCount = 0;
maxNum = 0;
maxNum = 0;
maxNumIndex = 0;
deficitAmount = 0;
holderB = 0;
int period = 0;
int offset_x = 0;
int[] Copy1 = new int[IntNumContracts + 1];
int[] Copy2 = new int[IntNumContracts + 1];
int[] Best2 = new int[IntNumContracts + 1];

for (i = 0; i < IntNumContracts + 1; i++)
{
 Copy1[i] = new Double(Pool[indexHolder][i]).intValue();
}
requiredRepair = 0;
while (numDeficit > 0)
{
 requiredRepair = 0;
 numDeficit = 0;
 turns = turns + 1;
 DeficitArray = new double[NumPeriods][IntNumContracts + 1];
 CommonContracts = new int[IntNumContracts + 1];

 for (i = 0; i < IntNumContracts; i++)
 {
  CommonContracts[i] = 0;
 }

 //--------------
 // This code finds the deficit amount for each period
 //--------------

 for (k = 1; k <= NumPeriods; k++)
 {
  temp3 = 0;
  if (k == 1)
  {
   index = k - 1;
```

```
          for (j = 1; j <= NumPeriods; j++)
          {
           index = index + 1;
           temp1 = 0;
           temp1 = Copy1[index];
           temp3 = temp3 + temp1;
          }
         }
         if (k > 1)
         {
          index = k - 1;
          offset_x = k - 2;
          for (i = 1; i <= k; i++)
          {
           for (j = k; j <= NumPeriods; j++)
           {
            index = index + 1;
            temp1 = 0;
            temp1 = Copy1[index];
            temp3 = temp3 + temp1;
           }
           index = index + offset_x;
           offset_x = offset_x - 1;
          }
         }
         if (temp3 < D[k - 1])
         {
          numDeficit = numDeficit + 1;
          requiredRepair = 1;
          difference = D[k - 1] - temp3;
          sizeDifference = new Double(Math.ceil(difference)).intValue();
          DeficitArray[k - 1][0] = sizeDifference;
         }
         else
         {
          DeficitArray[k - 1][0] = 0;
         }
        }

//-----------------------------
// This code fills in contracts for deficit periods
//------------------------

       for (k = 1; k <= NumPeriods; k++)
       {
        if (DeficitArray[k - 1][0] != 0)
```

```
 {
  if (k == 1)
  {
   countPeriod3 = 0;
   index = k - 1;

   //----------------------------------------
   // DeficitArray: One array for every period
   // first index in every array is the deficit for that period
   // Subsequent genes list the contracts which are active for a given period
   //----------------------------------------

   for (j = 1; j <= NumPeriods; j++)
   {
    index = index + 1;
    countPeriod3 = countPeriod3 + 1;
    DeficitArray[k - 1][countPeriod3] = index;
   }
  }
  if (k > 1)
  {
   countPeriod3 = 0;
   index = k - 1;
   offset_x = k - 2;
   for (i = 1; i <= k; i++)
   {

    //----------------------------------------
    // DeficitArray: One array for every period
    // first index in every array is the deficit for that period
    // Subsequent genes list the contracts which are active for a given period
    //----------------------------------------

    for (j = k; j <= NumPeriods; j++)
    {
     index = index + 1;
     countPeriod3 = countPeriod3 + 1;
     DeficitArray[k - 1][countPeriod3] = index;
    }
    index = index + offset_x;
    offset_x = offset_x - 1;
   }
  }
 }
}
```

```
    //-----------------------------------------------
    // This code counts, for each contract, the number of times
    //  it appears in a deficit period
    //-----------------------------------------------
    int holder2 = 0;
    for (k5 = 0; k5 < NumPeriods; k5++)
    {
     if (DeficitArray[k5][0] != 0)
     {
      for (k6 = 1; k6 < IntNumContracts + 1; k6++)
      {
       // Note: holder contains a contact #, which is an index in the CommonContracts
array
       holder = new Double(DeficitArray[k5][k6]).intValue();
       if (holder != 0)
       {
        ConCount = CommonContracts[holder];
        ConCount = ConCount + 1;
        CommonContracts[holder] = ConCount;
       }
      }
     }
    }

    //-----------------------------------------------
    // NumCommonContracts = variable containing the total number of
    // contracts in periods with deficits
    //-----------------------------------------

    CommonContracts[0] = 0;

    //-----------------------
    // The code that follows assigns the index number of the 1st contract to
    // have the highest number of occurances in a deficit period to variable
    // maxNumIndex; the highest number of occurances is stored in maxNum
    //--------------------------------------

    maxNum = 0;
    maxNumIndex = 0;
    for (k6 = 1; k6 <= IntNumContracts; k6++)
    {
     holder = CommonContracts[k6];
     if (holder >= maxNum)
     {
      maxNum = holder;
      maxNumIndex = k6;
```

```
    }
   }
   if (maxNumIndex > 0)
   {
    for (k5 = 0; k5 < NumPeriods; k5++)
    {
     if (DeficitArray[k5][0] != 0)
     {
      for (k6 = 1; k6 < IntNumContracts + 1; k6++)
      {
       if (maxNumIndex == DeficitArray[k5][k6])
       {
        period = k5;
        k6 = IntNumContracts + 1;
        k5 = NumPeriods;
       }
      }
     }
    }

    deficitAmount = new Double(DeficitArray[period][0]).intValue();
    DeficitArray[period][0] = 0;

    //----------------------
    // holderB = the deficitAmount for a given period divided by
    // the number of contracts of the given period
    //-----------------------

    holderB = new Integer(Rounding.toString(deficitAmount, 0)).intValue();
    holder = Copy1[maxNumIndex];
    holder = holder + holderB;
    Copy1[maxNumIndex] = holder;
   }
  }
  double Total;
  double cumulativeTotal;
  int index2;
  int index3;
  double size1;

  Total = 0;
  cumulativeTotal = 0;
  for (j = 1; j <= IntNumContracts; j++)
  {
   index2 = Contract_Period[j - 1];
   index3 = Contract_Length[j - 1];
```

```
  size1 = Copy1[j];
  if (size1 > 0)
  {
   /*********************************/
   TCQ[j - 1] = rp_Array[index2 - 1] * ( (B[0] * index3) + B[2]);
   TCX[j - 1] = rp_Array[index2 - 1] * B[1] * index3;
   Total = 0;
   Total = TCQ[j - 1] * size1 + TCX[j - 1];
   /*********************************/

   cumulativeTotal = cumulativeTotal + Total;
   Copy1[0] = new Double(cumulativeTotal).intValue();
  }
}

for (i = 0; i < IntNumContracts + 1; i++)
{
 OffSpring1[i] = new Integer(Copy1[i]).doubleValue();
}
temp1 = 0;
temp3 = 0;
index = 0;
difference = 0;
sizeDifference = 0;
feasibilityFlagPool = 0;

Total = 0;
cumulativeTotal = 0;

index3 = 0;
// ------------------------------------------------

for (i = 0; i < IntNumContracts + 1; i++)
{
 CommonContracts[i] = 0;
 for (j = 0; j < NumPeriods; j++)
 {
  DeficitArray[j][i] = 0;
 }
}

period = 0;
turns = turns + 1;
temp3=0;
for (i = 0; i < IntNumContracts + 1; i++)
{
```

```
  Copy2[i] = new Double(Pool[indexHolder][i]).intValue();
  Best2[i] = new Double(Pool[indexHolder][i]).intValue();
}
Best2[0]=9999999;
for(int alpha=0;alpha<20;alpha++)
{
  for (i = 0; i < NumPeriods; i++)
  {
    for (j = 0; j <= IntNumContracts; j++)
    {
      DeficitArray[i][j] = 0;
      CommonContracts[j] = 0;
    }
  }
  numDeficit = 1;
  while (numDeficit > 0)
  {
    requiredRepair = 0;
    numDeficit = 0;
    turns = turns + 1;
    DeficitArray = new double[NumPeriods][IntNumContracts + 1];
    //--------------
    // This code finds the deficit amount for each period
    //--------------

    for (k = 1; k <= NumPeriods; k++)
    {
      temp3 = 0;
      if (k == 1)
      {
        index = k - 1;
        for (j = 1; j <= NumPeriods; j++)
        {
          index = index + 1;
          temp1 = 0;
          temp1 = Copy2[index];
          temp3 = temp3 + temp1;
        }
      }
      if (k > 1)
      {
        index = k - 1;
        offset_x = k - 2;
        for (i = 1; i <= k; i++)
        {
          for (j = k; j <= NumPeriods; j++)
```

```
            {
             index = index + 1;
             temp1 = 0;
             temp1 = Copy2[index];
             temp3 = temp3 + temp1;
            }
            index = index + offset_x;
            offset_x = offset_x - 1;
           }
          }
          if (temp3 < D[k - 1])
          {
           numDeficit = numDeficit + 1;
           requiredRepair = 1;
           difference = D[k - 1] - temp3;
           sizeDifference = new Double(Math.ceil(difference)).intValue();
           DeficitArray[k - 1][0] = sizeDifference;
          }
          else
          {
           DeficitArray[k - 1][0] = 0;
          }
         }
         if (numDeficit == 0)
          break;
//-----------------------------
// This code fills in contracts for deficit periods
//------------------------

        for (k = 1; k <= NumPeriods; k++)
        {
         if (DeficitArray[k - 1][0] != 0)
         {
          if (k == 1)
          {
           countPeriod3 = 0;
           index = k - 1;

           //----------------------------------------
           // DeficitArray: One array for every period
           // first index in every array is the deficit for that period
           // Subsequent genes list the contracts which are active for a given period
           //----------------------------------------

           for (j = 1; j <= NumPeriods; j++)
           {
```

```
        index = index + 1;
        countPeriod3 = countPeriod3 + 1;
        DeficitArray[k - 1][countPeriod3] = index;
       }
      }
      if (k > 1)
      {
       countPeriod3 = 0;
       index = k - 1;
       offset_x = k - 2;
       for (i = 1; i <= k; i++)
       {

        //----------------------------------------
        // DeficitArray: One array for every period
        // first index in every array is the deficit for that period
        // Subsequent genes list the contracts which are active for a given period
        //----------------------------------------

        for (j = k; j <= NumPeriods; j++)
        {
         index = index + 1;
         countPeriod3 = countPeriod3 + 1;
         DeficitArray[k - 1][countPeriod3] = index;
        }
        index = index + offset_x;
        offset_x = offset_x - 1;
       }
      }
     }
    }
    for (int l = NumPeriods - 1; l >= 0; l--)
    {
     if (DeficitArray[l][0] != 0)
     {
      period = l;
      break;
     }
    }
    int counter = 0;
    for (int l = 1; l < DeficitArray[period].length; l++)
    {
     if (DeficitArray[period][l] == 0)
     {
      counter = l - 1;
      break;
```

```
    }
   }
   int rand =1+ this.Rand_Number.nextInt(counter-1);
   int contract = new Double(DeficitArray[period][rand]).intValue();
   deficitAmount = new Double(DeficitArray[period][0]).intValue();
   DeficitArray[period][0] = 0;

   //----------------------
   // holderB = the deficitAmount for a given period divided by
   // the number of contracts of the given period
   //-----------------------

   holderB = new Integer(Rounding.toString(deficitAmount, 0)).intValue();
   holder = Copy2[contract];
   holder = holder + holderB;
   Copy2[contract] = holder;
 }

 //--------------------
 Total = 0;
 cumulativeTotal = 0;
 for (j = 1; j <= IntNumContracts; j++)
 {
   index2 = Contract_Period[j - 1];
   index3 = Contract_Length[j - 1];
   size1 = Copy2[j];
   if (size1 > 0)
   {
     /*********************************/
     TCQ[j - 1] = rp_Array[index2 - 1] * ( (B[0] * index3) + B[2]);
     TCX[j - 1] = rp_Array[index2 - 1] * B[1] * index3;
     Total = 0;
     Total = TCQ[j - 1] * size1 + TCX[j - 1];
     /*********************************/

     cumulativeTotal = cumulativeTotal + Total;
     Copy2[0] = new Double(cumulativeTotal).intValue();
   }
 }
 if(Best2[0]>Copy2[0])
 {
   for(int beta=0;beta<this.IntNumContracts;beta++)
   {

     Best2[beta] = Copy2[beta];
   }
```

```
  }
  for(int m=0;m<this.IntNumContracts;m++)
  {
   Copy2[m]= new Double(Pool[indexHolder][m]).intValue();
  }
}
for(int beta=0;beta<this.IntNumContracts;beta++)
{
 Copy2[beta] = Best2[beta];
}

for (i = 0; i < IntNumContracts + 1; i++)
{
 OffSpring2[i] = new Integer(Copy2[i]).doubleValue();
}

temp1 = 0;
temp3 = 0;
index = 0;
difference = 0;
sizeDifference = 0;

if (OffSpring1[0] < OffSpring2[0])
{
 for (i = 0; i < IntNumContracts + 1; i++)
 {
  Pool[indexHolder][i] = OffSpring1[i];
 }
 Repair1 = Repair1 + 1;
}

if (OffSpring1[0] == OffSpring2[0])
{
 for (i = 0; i < IntNumContracts + 1; i++)
 {
  Pool[indexHolder][i] = OffSpring1[i];
 }
 RepairTie = RepairTie + 1;
}

if (OffSpring1[0] > OffSpring2[0])
{
 for (i = 0; i < IntNumContracts + 1; i++)
 {
  Pool[indexHolder][i] = OffSpring2[i];
 }
```

```
    Repair2 = Repair2 + 1;
  }
}

private void evaluation_function_pool(int NumPeriods)
{
  int i;
  int j;
  int size1;
  int Purchased = 0;
  double cumulativeTotal = 0;
  int index2;
  int index3;
  double Total = 0;
  int[] Copy = new int[IntNumContracts];

  NumContracts = .5 * NumPeriods * (NumPeriods + 1);
  IntNumContracts = new Double(NumContracts).intValue();

  TCQ = new double[IntNumContracts];
  TCX = new double[IntNumContracts];

  for (i = 1; i < IntNumContracts + 1; i++)
  {
    Copy[i - 1] = new Double(Pool[indexHolder][i]).intValue();
  }

  Total = 0;
  cumulativeTotal = 0;
  for (j = 0; j < IntNumContracts; j++)
  {
    index2 = Contract_Period[j];
    index3 = Contract_Length[j];
    size1 = Copy[j];
    if (size1 > 0)
    {
      TCQ[j] = rp_Array[index2 - 1] * ( (B[0] * index3) + B[2]);
      TCX[j] = rp_Array[index2 - 1] * B[1] * index3;

      Total = 0;
      Total = TCQ[j] * size1 + TCX[j];

      cumulativeTotal = cumulativeTotal + Total;
      Pool[indexHolder][0] = cumulativeTotal;
    }
  }
```

```
  for (i = 1; i < IntNumContracts + 1; i++)
  {
   Pool[indexHolder][i] = new Integer(Copy[i - 1]).doubleValue();
  }
}

private int record_best_chromosome_from_Pool(int NumPeriods)
{
  int i;
  int j;
  int x;
  int y;
  int count = 0;
  Vector Best = new Vector();
  Vector Test = new Vector();
  int same = 0;

  NumContracts = .5 * NumPeriods * (NumPeriods + 1);
  IntNumContracts = new Double(NumContracts).intValue();
  change = 0;

  if (gen == 0)
  {
   for (i = 0; i < NumOffspring + this.PopSize; i++)
   {
    if (i == 0)
    {
     for (j = 0; j < IntNumContracts + 1; j++)
     {
      bestChromosome[j] = Pool[i][j];
     }
    }
    if (i > 0)
    {
     if (bestChromosome[0] > Pool[i][0])
     {
      for (j = 0; j < IntNumContracts + 1; j++)
      {
       bestChromosome[j] = Pool[i][j];
      }
     }
     else if (bestChromosome[0] == Pool[i][0])
     {
      for (j = 0; j < IntNumContracts + 1; j++)
      {
       if (bestChromosome[j] != Pool[i][j])
```

```
          {
           same = 1;
           change = 1;
          }
         }
         if (same == 1)
         {
          for (j = 0; j < IntNumContracts + 1; j++)
          {
           bestChromosome[j] = Pool[i][j];
          }
         }
        }
       }
      }
     }
    else if (gen > 0)
    {
     for (i = 0; i < NumOffspring; i++)
     {
      if (bestChromosome[0] > Pool[i][0])
      {
       change = 1;
       for (j = 0; j < IntNumContracts + 1; j++)
       {
        bestChromosome[j] = Pool[i][j];
       }
      }
      else if (bestChromosome[0] == Pool[i][0])
      {
       for (j = 0; j < IntNumContracts + 1; j++)
       {
        if (bestChromosome[j] != Pool[i][j])
        {
         same = 1;
         change = 1;
        }
       }
       if (same == 1)
       {
        for (j = 0; j < IntNumContracts + 1; j++)
        {
         bestChromosome[j] = Pool[i][j];
        }
       }
      }
```

```
    }
   }
   return change;
}

public void write_to_file(boolean first_time, String FileName)
{
   int p = new Integer(Rounding.toString(gp * 10, 0)).intValue();
   int ad_d = new Integer(Rounding.toString(ad * 1, 0)).intValue();
   int d = new Integer(Rounding.toString(gd * 10, 0)).intValue();
   if (FileName.equals(""))
   {
     this.write = new WriteFile(buff, FileNum);
     if (first_time == true)
     {
       this.write.write_toFile();
     }
     else
     {
       this.write.append_toFile();
     }
     this.buff.delete(0, this.buff.length());
     this.buff = new StringBuffer();
   }
   else
   {
     this.write = new WriteFile(buff2, FileNum, FileName);
     if (first_time == true)
     {
       this.write.write_toFile();
     }
     else
     {
       this.write.append_toFile();
     }
     this.buff2.delete(0, this.buff2.length());
     this.buff2 = new StringBuffer();
   }
}

public void Fill_Best_Vector()
{
   int i;
   int j;
   int k;
   int Size;
```

```
double[] tempA;
double[] tempB;
NumContracts = .5 * NumPeriods * (NumPeriods + 1);
IntNumContracts = new Double(NumContracts).intValue();

tempA = new double[IntNumContracts + 1];
tempB = new double[IntNumContracts + 1];
int Size_of_Vector = 20;

if (gen == 0)
{
 for (k = 0; k < Size_of_Vector; k++)
 {
  tempB[0] = 999999999;
  Fill_Best.addElement(tempB);
 }
}
for (i = 0; i < (NumOffspring + PopSize); i++)
{
 for (j = Size_of_Vector - 1; j >= 0; j--)
 {
  tempA = (double[]) Fill_Best.elementAt(j);
  if (Pool[i][0] < tempA[0])
  {
   if (j != 0)
   {
    continue;
   }
   else
   {
    Fill_Best.insertElementAt( (double[]) Pool[i].clone(), 0);
    Fill_Best.removeElementAt(Fill_Best.size() - 1);
   }
  }
  else if (Pool[i][0] == tempA[0])
  {
   tempB = (double[]) Pool[i].clone();
   int p;
   for (p = 0; p < IntNumContracts + 1; p++)
   {
    if (tempA[p] != tempB[p])
    {
     p = IntNumContracts + 2;
     Fill_Best.insertElementAt( (double[]) Pool[i].clone(), j + 1);
     Size = Fill_Best.size();
     if (Size > Size_of_Vector)
```

```
            {
              Fill_Best.removeElementAt(Fill_Best.size() - 1);
            }
          }
        }
        break;
      }
      else if (Pool[i][0] > tempA[0])
      {
        if (Size_of_Vector - 1 == j)
        {
          Fill_Best.addElement( (double[]) Pool[i].clone());
        }
        else
        {
          Fill_Best.insertElementAt( (double[]) Pool[i].clone(), j + 1);
        }
        Size = Fill_Best.size();
        if (Size > Size_of_Vector)
        {
          Fill_Best.removeElementAt(Fill_Best.size() - 1);
        }
        break;
      }
    }
  }
  Fill_Best.trimToSize();
}

//If you send the timeperiod it will return the index number of all
// valid contracts in a Vector
public Vector valid_contracts(int timeperiod)
{
    Vector valid = new Vector();
    int x = this.NumPeriods;
    int y = timeperiod;
    for (int i =1;i<=timeperiod;i++)
    {
        for(int j=x;j>=y;j--)
        {
            valid.addElement(new Integer(j));
        }
        x= x+this.NumPeriods-i;
        y=y+this.NumPeriods -i;
    }
    return valid;
```

```
  }


/*************** MAIN ***************/
public static void main(String[] args)
{
  ea ea1 = new ea();
  System.out.println("Begin EA");

  ea1.NumPeriods = 6;
  ea1.PopSize = 40;
  ea1.maxGenerations = 5000;
  int n = 0;
  int z;
  int ProblemNumber = 0;
  int x;
  int y;
  int count = 0;
  int term = 0;
  double Term_Condition;
  double prob_purchase;
  double randNum;
  int num1;
  int j, k;
  double number;
  int number2;
  int number3;
  double tempz;
  double tempzz;
  double tempy;
  double Restart = 0;
  int Restart_Counter = 0;
  int Gen_Best_Found = 0;
  long TimetoComplete = 0;
  long TimetoBest = 0;
  int Size = 1;
  double increment_gp = .1;
  double increment_gd = .1;
  int increment_n = 6;
  int increment_Size = 1;
  int x_count = 0;
  int y_count = 0;
  int z_count = 0;
  String econScale = "x";
  ea1.n_scale = 0;
  int tempRepair1 = 0;
```

```
int tempRepair2 = 0;
int tempRepairTie = 0;
ea1.gp = .10;
ea1.ad = 540;
ea1.gd = .10;
ea1.Date = 10.26;
int i;
int runNumber;
int Bingo = 0;
int numRuns;
int remainder;
int numProb;
int AdjPopSize;
int AdjMaxGen;
int AdjInitialDemand;
int incrementPrice = 0;

ea1.UniformCrossoverRate = .8;
ea1.OnePointCrossoverRate = .2;
ea1.MutationRate2 = .6;
ea1.MutationRate1 = .6;

ea1.NumPeriods = 0;
for (int x_n = 1; x_n <= 3; x_n++)
{
 int buff_Length = ea1.buff.length();
 ea1.buff.delete(0, buff_Length);
 ea1.buff = new StringBuffer();
 ea1.NumPeriods = ea1.NumPeriods + increment_n;
 ea1.n_scale = ea1.n_scale + 1;
 ea1.NumContracts = .5 * ea1.NumPeriods * (ea1.NumPeriods + 1);
 ea1.IntNumContracts = new Double(ea1.NumContracts).intValue();
 ea1.PoolSize = ea1.PopSize * 200;
 ea1.Best_All_Runs = new double[ea1.IntNumContracts + 1];
 ea1.bestChromosome = new double[ea1.IntNumContracts + 1];
 double[] overall_Best = new double[ea1.IntNumContracts + 1];
 int[] OffSpring1 = new int[ea1.IntNumContracts];
 int[] OffSpring2 = new int[ea1.IntNumContracts];
 Size = 0;
 y_count = 0;

 for (int x_Size = 1; x_Size <= 3; x_Size++)
 {
  ea1.gen = 0;
  Size = Size + 1;
  y_count = 0;
```

```
ea1.gp = .1;
ea1.gd = .1;

buff_Length = ea1.buff.length();
ea1.buff.delete(0, buff_Length);
ea1.buff = new StringBuffer();

if (Size == 1)
{
 //Small economies of scale
 /* */
 ea1.B1 = 13.116;
 ea1.B2 = 962.927;
 ea1.B3 = 258.526;
 econScale = "econScale-Small";
 /* */
}
if (Size == 2)
{
 //Medium economies of scale
 /* */
 ea1.B1 = 6.757;
 ea1.B2 = 928.503;
 ea1.B3 = 364.989;
 econScale = "econScale-Medium";
 /* */
}
if (Size == 3)
{
 //Large economies of scale
 /* */
 ea1.B1 = 0.3977;
 ea1.B2 = 894.115;
 ea1.B3 = 471.416;
 econScale = "econScale-Large";
 /* */
}

ea1.B[0] = ea1.B1;
ea1.B[1] = ea1.B2;
ea1.B[2] = ea1.B3;

for (numProb = 1; numProb <= 9; numProb++)
{
 ProblemNumber = ProblemNumber + 1;
```

```
y_count = y_count + 1;
ea1.gen = 0;

ea1.NumContracts = .5 * ea1.NumPeriods * (ea1.NumPeriods + 1);
ea1.IntNumContracts = new Double(ea1.NumContracts).intValue();

ea1.fill_arrays();

if (ea1.NumPeriods == 6)
{
 if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 1)
 {
  ea1.iLP_Opt = 268496.3;

 }
 if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 2)
 {
  ea1.iLP_Opt = 325790.3;

 }
 if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 3)
 {
  ea1.iLP_Opt = 383054.8;
 }
 if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 1)
 {
  ea1.iLP_Opt = 368073.5;
 }
 if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 2)
 {
  ea1.iLP_Opt = 454755.3;
 }
 if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 3)
 {
  ea1.iLP_Opt = 541396.1;
 }
 if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 1)
 {
  ea1.iLP_Opt = 499789.2;
 }
 if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 2)
 {
  ea1.iLP_Opt = 627457.3;
 }
 if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 3)
 {
```

```
 ea1.iLP_Opt = 755068.4;
}
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 1)
{
 ea1.iLP_Opt = 248062.5;
}
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 2)
{
 ea1.iLP_Opt = 299026.1;
}
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 3)
{
 ea1.iLP_Opt = 349963.3;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 1)
{
 ea1.iLP_Opt = 316077.6;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 2)
{
 ea1.iLP_Opt = 387101.1;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 3)
{
 ea1.iLP_Opt = 458091.6;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 1)
{
 ea1.iLP_Opt = 405209.5;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 2)
{
 ea1.iLP_Opt = 503271.7;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 3)
{
 ea1.iLP_Opt = 601292.7;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 1)
{
 ea1.iLP_Opt = 231313.5;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 2)
{
 ea1.iLP_Opt = 277851.5;
}
```

```
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 3)
  {
   ea1.iLP_Opt = 324363.5;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 1)
  {
   ea1.iLP_Opt = 277942.6;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 2)
  {
   ea1.iLP_Opt = 337816.1;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 3)
  {
   ea1.iLP_Opt = 397656.8;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
      Size == 1)
  {
   ea1.iLP_Opt = 337080.9;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
      Size == 2)
  {
   ea1.iLP_Opt = 414361.3;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
      Size == 3)
  {
   ea1.iLP_Opt = 491600.5;
  }
 }
 if (ea1.NumPeriods == 12)
 {
  if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 1)
  {
   ea1.iLP_Opt = 438512.7;
  }
  if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 2)
  {
   ea1.iLP_Opt = 487799.3;
  }
  if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 3)
  {
   ea1.iLP_Opt = 537047.3;
  }
```

```
if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 1)
{
 ea1.iLP_Opt = 803822.1;
}
if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 2)
{
 ea1.iLP_Opt = 936621.8;
}
if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 3)
{
 ea1.iLP_Opt = 1069332.;
}
if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 1)
{
 ea1.iLP_Opt = 1536695.;
}
if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 2)
{
 ea1.iLP_Opt = 1859647.;
}
if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 3)
{
 ea1.iLP_Opt = 2182392.;
}
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 1)
{
 ea1.iLP_Opt = 351317.8;
}
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 2)
{
 ea1.iLP_Opt = 388381.6;
}
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 3)
{
 ea1.iLP_Opt = 420306.0;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 1)
{
 ea1.iLP_Opt = 526002.2;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 2)
{
 ea1.iLP_Opt = 598458.9;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 3)
{
```

```
 ea1.iLP_Opt = 664641.3;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 1)
{
 ea1.iLP_Opt = 839560.6;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 2)
{
 ea1.iLP_Opt = 983596.4;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 3)
{
 ea1.iLP_Opt = 1122124.;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 1)
{
 ea1.iLP_Opt = 276257.7;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 2)
{
 ea1.iLP_Opt = 325630.2;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 3)
{
 ea1.iLP_Opt = 358465.1;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 1)
{
 ea1.iLP_Opt = 363260.8;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 2)
{
 ea1.iLP_Opt = 429939.2;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 3)
{
 ea1.iLP_Opt = 477405.9;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
   Size == 1)
{
 ea1.iLP_Opt = 501702.1;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
   Size == 2)
{
```

```
    ea1.iLP_Opt = 597692.8;
   }
   if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
      Size == 3)
   {
    ea1.iLP_Opt = 671895.9;
   }
  }
  if (ea1.NumPeriods == 18)
  {
   if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 1)
   {
    ea1.iLP_Opt = 628844.50;
   }
   if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 2)
   {
    ea1.iLP_Opt = 657505.92;
   }
   if (ea1.gp == 0.1 && ea1.gd == 0.1 && Size == 3)
   {
    ea1.iLP_Opt = 686122.87;
   }
   if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 1)
   {
    ea1.iLP_Opt = 1513748.34;
   }
   if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 2)
   {
    ea1.iLP_Opt = 1704381.20;
   }
   if (ea1.gp == 0.1 && ea1.gd == 0.2 && Size == 3)
   {
    ea1.iLP_Opt = 1894885.93;
   }
   if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 1)
   {
    ea1.iLP_Opt = 4187309.76;
   }
   if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 2)
   {
    ea1.iLP_Opt = 4993621.49;
   }
   if (ea1.gp == 0.1 && ea1.gd == 0.30000000000000004 && Size == 3)
   {
    ea1.iLP_Opt = 5799518.77;
   }
```

```
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 1)
{
 ea1.iLP_Opt = 391171.05;
}
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 2)
{
 ea1.iLP_Opt = 440385.34;
}
if (ea1.gp == 0.2 && ea1.gd == 0.1 && Size == 3)
{
 ea1.iLP_Opt = 462806.17;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 1)
{
 ea1.iLP_Opt = 675913.15;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 2)
{
 ea1.iLP_Opt = 771270.01;
}
if (ea1.gp == 0.2 && ea1.gd == 0.2 && Size == 3)
{
 ea1.iLP_Opt = 833514.47;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 1)
{
 ea1.iLP_Opt = 1365808.49;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 2)
{
 ea1.iLP_Opt = 1593712.59;
}
if (ea1.gp == 0.2 && ea1.gd == 0.30000000000000004 && Size == 3)
{
 ea1.iLP_Opt = 1779087.80;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 1)
{
 ea1.iLP_Opt = 285071.26;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 2)
{
 ea1.iLP_Opt = 333592.52;
}
if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.1 && Size == 3)
{
```

```
   ea1.iLP_Opt = 368648.84;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 1)
  {
   ea1.iLP_Opt = 391247.28;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 2)
  {
   ea1.iLP_Opt = 458638.36;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.2 && Size == 3)
  {
   ea1.iLP_Opt = 508518.68;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
      Size == 1)
  {
   ea1.iLP_Opt = 590782.49;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
      Size == 2)
  {
   ea1.iLP_Opt = 695741.19;
  }
  if (ea1.gp == 0.30000000000000004 && ea1.gd == 0.30000000000000004 &&
      Size == 3)
  {
   ea1.iLP_Opt = 775928.02;
  }
 }
 ea1.FileNum = ea1.FileNum + 1;

 runNumber = 0;
 int MaxRuns = 5;
 double overall_BestofRuns = 0;
 double TimetoBestofRuns = 0;
 double TimetoCompleteforAllRuns = 0;
 int Gen_BestofRuns = 0;
 double[][] runFitnessTimes = new double[MaxRuns][5];
 double TotalTimetoCompleteforAllRuns = 0;

 System.out.println("\n");
 ea1.buff.append("\n");
 System.out.println("N = " + ea1.NumPeriods + ", gp = " + ea1.gp +
                    ", ad = " + ea1.ad +
                    ", gd = " + ea1.gd + ", B1 = " + ea1.B1 +
```

```
                ", B2 = " + ea1.B2 + ", B3 = " + ea1.B3 +
                ", LP Solution = " + ea1.iLP_Opt +
                ", # best the same last value = " + count +
                ", PopSize = " + ea1.PopSize +
                ", MaxGeneration = " + ea1.maxGenerations);
ea1.temp = "N = " + ea1.NumPeriods + ", gp = " + ea1.gp +
   ", ad = " + ea1.ad +
   ", gd = " + ea1.gd + ", B1 = " + ea1.B1 + " , B2 = " +
   ea1.B2 +
   ", B3 = " + ea1.B3 + ", LP Solution = " + ea1.iLP_Opt +
   ", # best the same last value = " + count +
   ", PopSize = " + ea1.PopSize +
   ", MaxGeneration = " + ea1.maxGenerations;
ea1.buff.append(ea1.temp + "\r\n");

System.out.println(econScale + ", UniformCrossoverRate = " +
                ea1.UniformCrossoverRate +
                ", OnePointCrossoverRate = " +
                ea1.OnePointCrossoverRate +
                ", MutationRate2 = " + ea1.MutationRate2 +
                ", ea1.MutationRate1 = " + ea1.MutationRate1);
ea1.temp = econScale + ", UniformCrossoverRate = " +
   ea1.UniformCrossoverRate +
   ", OnePointCrossoverRate = " + ea1.OnePointCrossoverRate +
   ", MutationRate2 = " + ea1.MutationRate2 +
   ", ea1.MutationRate1 = " + ea1.MutationRate1;
ea1.buff.append(ea1.temp + "\r\n");

System.out.println("PopSize = " + ea1.PopSize +
                ", MaxGeneration = " + ea1.maxGenerations +
                ", # Restarts = " + Restart_Counter);
ea1.temp = "PopSize = " + ea1.PopSize + ", MaxGeneration = " +
   ea1.maxGenerations;
ea1.buff.append(ea1.temp + "\r\n");

for (numRuns = 1; numRuns <= MaxRuns; numRuns++)
{
 ea1.gen = 0;

 ea1.initialize_chromosomes();
 long Start = System.currentTimeMillis();

 TimetoComplete = 0;
 TimetoBest = 0;
 term = 0;
 Restart = 0;
```

```
            Restart_Counter = 0;
            overall_Best[0] = 999999999;
            Gen_Best_Found = 0;
            count = 0;
            runNumber = runNumber + 1;

            System.out.println(" ");
            ea1.buff.append("\n");
            System.out.println("Problem Number = " + ProblemNumber +
                               ", Run Number = " + runNumber);
            ea1.temp = "Problem Number = " + ProblemNumber +
               ", Run Number = " + runNumber;
            ea1.buff.append(ea1.temp + "\r\n");
            ea1.counter = ea1.PopSize;

            for (ea1.gen = 0; ea1.gen < ea1.maxGenerations; ea1.gen++)
            {
             ea1.NumOffspring = ea1.PopSize;
             ea1.counter = ea1.PopSize;
             ea1.initialize_chromosome_pool(ea1.PopSize, ea1.IntNumContracts);

             ea1.mutation_operator();
             ea1.OnePoint_crossover_operator();
             ea1.Uniform_crossover_operator();

             ea1.Repair1 = 0;
             ea1.Repair2 = 0;
             ea1.RepairTie = 0;
             for (ea1.indexHolder = 0;
                 ea1.indexHolder < (ea1.NumOffspring + ea1.PopSize);
                 ea1.indexHolder++)
             {
              ea1.repair_function_pool(ea1.NumPeriods);
              tempRepair1 = tempRepair1 + ea1.Repair1;
              tempRepair2 = tempRepair2 + ea1.Repair2;
              tempRepairTie = tempRepairTie + ea1.RepairTie;
             }

             ea1.change = ea1.record_best_chromosome_from_Pool(ea1.NumPeriods);

//Pool of best 20 chromosomes found throughout each run of an EA

             ea1.Fill_Best_Vector();
             if (ea1.change == 0)
             {
              count = count + 1;
```

```
  }
  else if (ea1.change == 1)
  {
   count = 0;
  }
  term = ea1.gen;
  if (overall_Best[0] > ea1.bestChromosome[0])
  {
   Gen_Best_Found = ea1.gen;
   for (i = 0; i < ea1.IntNumContracts + 1; i++)
   {
    overall_Best[i] = ea1.bestChromosome[i];
   }
   long ToBest = System.currentTimeMillis();
   TimetoBest = ToBest - Start;
  }
  for (i = 0; i < ea1.PopSize; i++)
  {
   for (j = 0; j < ea1.IntNumContracts + 1; j++)
   {
    ea1.Chromosome[i][j] = 0;
   }
  }
  for (ea1.indexHolder = 0;
      ea1.indexHolder < ea1.IntNumContracts + 1;
      ea1.indexHolder++)
  {
   ea1.Chromosome[0][ea1.indexHolder] =
       ea1.bestChromosome[ea1.indexHolder];
  }

  ea1.select_next_generation();
  for (i = 0; i < ea1.PoolSize; i++)
  {
   for (j = 0; j < ea1.IntNumContracts + 1; j++)
   {
    ea1.Pool[i][j] = 0;
   }
  }
  if (count >= 100 * ea1.NumPeriods)
  {
   term = ea1.gen;
   ea1.gen = ea1.maxGenerations;
  }
 }
```

```
System.out.println("Number of times Repair #1 best = " +
                tempRepair1);
System.out.println("Number of times Repair #2 best = " +
                tempRepair2);
System.out.println(
   "Number of times Repair #1 and Repair #2 Tied = " +
   tempRepairTie);
ea1.temp = "Number of times Repair #1 best = " +
   tempRepair1;
ea1.buff.append(ea1.temp + "\r\n");
ea1.temp = "Number of times Repair #2 best = " +
   tempRepair2;
ea1.buff.append(ea1.temp + "\r\n");
ea1.temp = "Number of times Repair #1 and Repair #2 Tied = " +
   tempRepairTie;
ea1.buff.append(ea1.temp + "\r\n");

long End = System.currentTimeMillis();
TimetoComplete = End - Start;
System.out.println("The End!");
ea1.temp = "The End!";
ea1.buff.append(ea1.temp + "\r\n");

runFitnessTimes[numRuns - 1][0] = overall_Best[0];
runFitnessTimes[numRuns - 1][1] = Gen_Best_Found;
runFitnessTimes[numRuns - 1][2] = TimetoBest;
runFitnessTimes[numRuns - 1][3] = TimetoComplete;

if (numRuns == 1)
{
 overall_BestofRuns = runFitnessTimes[numRuns - 1][0];
 Gen_BestofRuns = new Double(runFitnessTimes[numRuns - 1][1]).
    intValue();
 TotalTimetoCompleteforAllRuns = TimetoComplete;
 TimetoBestofRuns = runFitnessTimes[numRuns - 1][2];

 for (z = 0; z < ea1.IntNumContracts + 1; z++)
 {
  ea1.Best_All_Runs[z] = ea1.bestChromosome[z];
 }
}
else
{
 if (runFitnessTimes[numRuns - 2][0] >
    runFitnessTimes[numRuns - 1][0])
 {
```

```
   overall_BestofRuns = runFitnessTimes[numRuns - 1][0];
   Gen_BestofRuns = new Double(runFitnessTimes[numRuns - 1][1]).
      intValue();
   TimetoBestofRuns = runFitnessTimes[numRuns - 1][2];
   for (j = 0; j < ea1.IntNumContracts + 1; j++)
   {
    ea1.Best_All_Runs[j] = ea1.bestChromosome[j];
   }
  }
  else
  {
   if (runFitnessTimes[numRuns - 2][0] ==
      runFitnessTimes[numRuns - 1][0] &&
      runFitnessTimes[numRuns - 2][1] >
      runFitnessTimes[numRuns - 1][1])
   {
    overall_BestofRuns = runFitnessTimes[numRuns - 1][0];
    Gen_BestofRuns = new Double(runFitnessTimes[numRuns - 1][1]).
       intValue();
    TimetoBestofRuns = runFitnessTimes[numRuns - 1][2];
    for (j = 0; j < ea1.IntNumContracts + 1; j++)
    {
     ea1.Best_All_Runs[j] = ea1.bestChromosome[j];
    }
   }
  }
  TotalTimetoCompleteforAllRuns = TotalTimetoCompleteforAllRuns +
     TimetoComplete;
}
System.out.println("Gen # Run terminated = " + term);
ea1.temp = "Gen # Run terminated = " + term;
ea1.buff.append(ea1.temp + "\r\n");
System.out.println("Gen to find Best Chromosome this run = " +
                runFitnessTimes[numRuns - 1][1]);
ea1.temp = "Gen to find Best Chromosome this run = " +
   runFitnessTimes[numRuns - 1][1];
ea1.buff.append(ea1.temp + "\r\n");
System.out.println("Time to find Best Chromosome = " +
                runFitnessTimes[numRuns - 1][2] +
                " milliseconds; i.e., " +
                (runFitnessTimes[numRuns - 1][2] / 1000) +
                " seconds");
ea1.temp = "Time to find Best Chromosome = " +
   runFitnessTimes[numRuns - 1][2] + " milliseconds; i.e., " +
   (runFitnessTimes[numRuns - 1][2] / 1000) + " seconds";
ea1.buff.append(ea1.temp + "\r\n");
```

```
            System.out.println("Time to Complete EA run = " +
                        runFitnessTimes[numRuns - 1][3] +
                        " milliseconds; i.e., " +
                        (runFitnessTimes[numRuns - 1][3] / 1000) +
                        " seconds");
            ea1.temp = "Time to Complete EA run = " +
               runFitnessTimes[numRuns - 1][3] + " milliseconds; i.e., " +
               (runFitnessTimes[numRuns - 1][3] / 1000) + " seconds";
            ea1.buff.append(ea1.temp + "\r\n");
            System.out.println(" ");
            System.out.println("EA Solution - LP Solution = " +
                        (overall_BestofRuns -
                         ea1.iLP_Opt));
            ea1.temp = "EA Solution - LP Solution = " +
               (overall_BestofRuns - ea1.iLP_Opt);
            ea1.buff.append(ea1.temp + "\r\n");
            System.out.println("Precent Error = " +
                        ( ( (overall_BestofRuns -
                            ea1.iLP_Opt) / ea1.iLP_Opt) * 100));
            ea1.temp = "Precent Error = " +
               ( ( (overall_BestofRuns - ea1.iLP_Opt) /
                  ea1.iLP_Opt) * 100);
            ea1.buff.append(ea1.temp + "\r\n\r\n");
            runFitnessTimes[numRuns -
               1][4] = ( ( (overall_BestofRuns - ea1.iLP_Opt) / ea1.iLP_Opt) *
                     100);
            System.out.println(" ");
            ea1.buff.append("\r\n");
            System.out.print("Run #" + numRuns + ". ");
            ea1.temp = "Run #" + numRuns + "). ";
            ea1.buff.append(ea1.temp);
            for (y = 0; y < 5; y++)
            {
             System.out.print(runFitnessTimes[numRuns - 1][y] + " | ");
             ea1.temp = runFitnessTimes[numRuns - 1][y] + " | ";
             ea1.buff.append(ea1.temp);
            }
            System.out.println(" ");
            ea1.buff.append("\r\n");
            System.out.println("(1: best fitness value, 2: gen best found, 3: time to best
(msec), 4: time to complete (msec), 5:% error)");
            ea1.buff.append("\r\n");
            System.out.println(" ");
            ea1.buff.append("\r\n");
            System.out.print("Best All Runs: ");
            ea1.temp = "Best All Runs: ";
```

```
  ea1.buff.append(ea1.temp + "\r\n");
  System.out.println("Best Chromosome for Run Number #" + numRuns +
                     ": ");
  ea1.temp = "Best Chromosome for Run Number #" + numRuns + ": ";
  ea1.buff.append(ea1.temp + "\r\n");
  for (j = 0; j < ea1.IntNumContracts + 1; j++)
  {
   System.out.print(ea1.bestChromosome[j] + " | ");
   ea1.temp = ea1.bestChromosome[j] + " | ";
   ea1.buff.append(ea1.temp);
  }
  System.out.println(" ");
  ea1.buff.append("\r\n");
  System.out.println("Best 20 chromosomes from EA run #" + numRuns +
                     " for Problem # " + ea1.FileNum);
  ea1.buff2.append("Best 20 chromosomes from EA run  \r\n");

  double[] tempA;

  for (i = 0; i < 20; i++)
  {
   tempA = (double[]) ea1.Fill_Best.elementAt(i);
   for (j = 0; j < tempA.length; j++)
   {
    ea1.temp2 = new Double(tempA[j]).toString() + " | ";
    ea1.buff2.append(ea1.temp2);
   }
   ea1.buff2.append("\r\n");
  }
  ea1.buff2.append("\r\n");
  ea1.Fill_Best.removeAllElements();
  if (numRuns == 1)
  {
   ea1.write_to_file(true, "Best Chromosomes ");
  }
  else
  {
   ea1.write_to_file(false, "Best Chromosomes ");
  }
  System.out.println(" ");
  ea1.buff.append("\r\n");
 }
 System.out.println(" ");
 ea1.buff.append("\r\n");
 System.out.println("Total Time to Complete " + MaxRuns +
                    " EA runs = " +
```

```
                TimetoCompleteforAllRuns +
                " milliseconds; i.e., " +
                (TimetoCompleteforAllRuns / 1000) + " seconds");
ea1.temp = "Total Time to Complete " + MaxRuns + " EA runs = " +
   TimetoCompleteforAllRuns + " milliseconds; i.e., " +
   (TimetoCompleteforAllRuns / 1000) + " seconds";
ea1.buff.append(ea1.temp + "\r\n");
System.out.println("Average Total Time to Complete " + MaxRuns +
                " EA runs = " +
                (TimetoCompleteforAllRuns / MaxRuns) +
                " milliseconds; i.e., " +
                (TimetoCompleteforAllRuns / MaxRuns / 1000) +
                " seconds");
ea1.temp = "Average Total Time to Complete " + MaxRuns +
   " EA runs = " +
   (TimetoCompleteforAllRuns / MaxRuns) + " milliseconds; i.e., " +
   (TimetoCompleteforAllRuns / MaxRuns / 1000) + " seconds";
ea1.buff.append(ea1.temp + "\r\n");
System.out.println("Best Fitness Value found during " + MaxRuns +
                " EA runs = " +
                overall_BestofRuns);
ea1.temp = "Best Fitness Value found during " + MaxRuns +
   " EA runs = " +
   overall_BestofRuns;
ea1.buff.append(ea1.temp + "\r\n");

for (x = 1; x <= MaxRuns; x++)
{
 System.out.print("Run #" + x + ". ");
 ea1.temp = "Run #" + x + "). ";
 ea1.buff.append(ea1.temp + "\r\n");
 for (y = 0; y < 5; y++)
 {
  System.out.print(runFitnessTimes[x - 1][y] + " | ");
  ea1.temp = runFitnessTimes[x - 1][y] + " | ";
  ea1.buff.append(ea1.temp);
 }
 System.out.println(" ");
 ea1.buff.append("\r\n");
}
System.out.println(" ");
ea1.buff.append("\r\n");
System.out.print("Best All Runs: ");
ea1.temp = "Best All Runs: ";
ea1.buff.append(ea1.temp + "\r\n");
for (j = 0; j < ea1.IntNumContracts + 1; j++)
```

```
{
 System.out.print(ea1.Best_All_Runs[j] + " | ");
 ea1.temp = ea1.Best_All_Runs[j] + " | ";
 ea1.buff.append(ea1.temp);
}
System.out.println("\n ");
ea1.buff.append("\r\n\n");
System.out.println("Total Time to Complete All " + numRuns + "Runs: " +
                TotalTimetoCompleteforAllRuns);
ea1.temp = "Total Time to Complete All " + numRuns + "Runs: " +
   TotalTimetoCompleteforAllRuns;
ea1.buff.append("\r\n");
System.out.println("D Array: ");
ea1.buff.append("D Array: ");
ea1.buff.append("\r\n");
for (j = 0; j < ea1.NumPeriods; j++)
{
 System.out.println(ea1.D[j]);
 ea1.temp = ea1.D[j] + "\n";
 ea1.buff.append(ea1.temp + "\r\n");
}
System.out.println("\n ");
ea1.buff.append("\r\n");
if (numRuns == 1)
{
 ea1.write_to_file(true, "");
}
else
{
 ea1.write_to_file(false, "");
}
Bingo = 0;

if (ea1.gd >= .3)
{
 if (z_count <= 2)
 {
  ea1.gp = ea1.gp + increment_gp;
  ea1.gd = 0;
  z_count = z_count + 1;
 }
 if (z_count > 2)
 {
  z_count = 0;
  ea1.gp = .1;
  ea1.gd = 0;
```

```
            }
          }
          ea1.gd = ea1.gd + increment_gd;
        }
      }
    }
  }

  private void jbInit()
      throws Exception
  {
  }
}
```

APPENDIX B: SOURCE CODE FOR SBCP-EA

Training

Start.java

```
package sim_ea;
import java.util.*;
import java.lang.*;
import java.lang.Math.*;

public class start
{
   /** initialize variables **/
   /************SIM VARIABLES*************************************/
   public sim sim1;
   public double sim_demand[][];
   public double sim_price[][];
   public double[][] sim_chromosome_value;
   public int sim_instances;

/*********************************************************************/
   int NumPeriods;
   int PopSize;
   int maxGenerations;
   double OnePointCrossoverRate;
   double UniformCrossoverRate;
   double MutationRate1;
   double MutationRate2;
   int gen;
   double[][] Chromosome;
   Random Rand_Number;
   ReadFile read;
   WriteFile write;
   double NumContracts;
   int n_scale;
   int IntNumContracts;
   double[] B = new double[3];
   String temp;
   String temp2;
   StringBuffer buff;
   StringBuffer buff2;
   double rp;
   double rd;
   double gp;
```

```
double ad;
double gd;
double B1;
double B2;
double B3;
double iLP_Opt;
double[] rp_Array;
double[] rd_Array;
double[] D;
int[] Contract_Length;
double[] TCQ;
double[] TCX;
int[] Contract_Period;
int[] NumContracts_EachPeriod;
int feasibilityFlag = 0;
int feasibilityFlagPool = 0;
int indexHolder;
double[] bestChromosome;
double[][] Pool;
int PoolSize;
int counter;
int NumOffspring;
double Date;
int FileNum = 0;
int change;
double[] Best_All_Runs;
int Repair1;
int Repair2;
int RepairTie;
int AlreadyDone = 0;
Vector Fill_Best = new Vector();

public start()
{
  this.Rand_Number = new Random();
  this.read = new ReadFile("input.txt");
  this.buff = new StringBuffer();
  this.buff2 = new StringBuffer();
}

private void read_input_file()
{
  System.out.println("DATE: " + Date);
  temp = "DATE: " + Date;
  buff.append(temp + "\r\n");
  System.out.println("N = " + NumPeriods + ", gp = " + gp + ", ad = " + ad +
```

```
                ", gd = " + gd + ", B1 = " + B1 + " , B2 = " + B2 +
                ", B3 = " + B3);
  temp = "N = " + NumPeriods + ", gp = " + gp + ", ad = " + ad + ", gd = " +
     gd + ", B1 = " + B1 + " , B2 = " + B2 + ", B3 = " + B3;
  buff.append(temp + "\r\n");

  NumContracts = .5 * NumPeriods * (NumPeriods + 1);
  IntNumContracts = new Double(NumContracts).intValue();

  System.out.println("Number of contracts =" + IntNumContracts +
                ", NumPeriods = " + NumPeriods);
  temp = "!Number of contracts =" + IntNumContracts;
  buff.append(temp + "\r\n");

  System.out.println("LP Optimal =" + iLP_Opt);
  temp = "LP Optimal =" + iLP_Opt;
  buff.append(temp + "\r\n");
}

private void fill_arrays()
{
  double product = 1;
  int count;
  int i;
  int j;
  int index1;
  int enumerate1;
  int enumerate2;
  Contract_Period = new int[IntNumContracts];
  Contract_Length = new int[IntNumContracts];
  Pool = new double[PoolSize][IntNumContracts + 1];
  int p = new Integer(Rounding.toString(gp * 10, 0)).intValue();
  int ad_d = new Integer(Rounding.toString(ad * 1, 0)).intValue();
  int d = new Integer(Rounding.toString(gd * 10, 0)).intValue();
  rp = 1 - gp;
  rd = 1 + gd;
  rp_Array = new double[NumPeriods];
  rd_Array = new double[NumPeriods];
  D = new double[NumPeriods];
  NumContracts_EachPeriod = new int[NumPeriods];

  rp_Array[0] = 1;
  for (count = 1; count < NumPeriods; count++)
  {
    product = product * rp;
    rp_Array[count] = product;
```

```
  }
  product = 1;
  rd_Array[0] = 1;
  for (count = 1; count < NumPeriods; count++)
  {
   product = product * rd;
   rd_Array[count] = product;
  }
  D[0] = ad;
  for (count = 1; count < NumPeriods; count++)
  {
   D[count] = new Double(Math.ceil(ad * rd_Array[count])).intValue();
  }
  for (i = 0; i < NumPeriods; i++)
  {
   NumContracts_EachPeriod[i] = (NumPeriods - (i)) * (i + 1);
  }
  index1 = 0;
  for (enumerate1 = 0; enumerate1 <= NumPeriods; enumerate1++)
  {
   for (enumerate2 = 1; enumerate2 <= NumPeriods - enumerate1;
       enumerate2++)
   {
    Contract_Length[index1] = enumerate2;
    index1 = index1 + 1;
   }
  }
  count = 0;
  for (i = 1; i < NumPeriods + 1; i++)
  {
   for (j = 1; j < NumPeriods + 2 - i; j++)
   {
    Contract_Period[count] = i;
    count = count + 1;
   }
  }
}

private void initialize_chromosomes()
{
  double prob_purchase = .5;
  double Num_Contracts = NumPeriods * .5 * (NumPeriods + 1);
  Chromosome = new double[PopSize][IntNumContracts + 1];
  int j;
  int k;
  double number;
```

```
  double tempz;
  double tempzz;
  double tempy;
  int number2;
  int x;
  int y;

  for (k = 0; k < PopSize; k++)
  {
   Chromosome[k][0] = 999999;
   for (j = 1; j <= IntNumContracts; j++)
   {
    number = Rand_Number.nextDouble();
    if (number <= prob_purchase)
    {
     tempz = Rand_Number.nextDouble() * 100;
     tempzz = Rand_Number.nextDouble() * 10;
     tempy = (tempz * tempzz) / 2;
     number2 = new Integer(Rounding.toString(tempy, 0)).intValue();
     Chromosome[k][j] = number2;
    }
    else
    {
     Chromosome[k][j] = 0;
    }
   }
  }
}

public void print_Chromosome()
{
  int start;
  int start1;
  System.out.println("Print Pool");
  for (start = 0; start < PopSize; start++)
  {
   for (start1 = 0; start1 < IntNumContracts; start1++)
   {
    System.out.print(Chromosome[start][start1] + " , ");
   }
   System.out.println();
  }
  System.out.println();

}
```

```
public void print_Chromosome(int indexHolder)
{
 System.out.println("Repaired Chromosome: ");
 for (int k5 = 0; k5 < this.IntNumContracts; k5++)
 {
  System.out.print(this.Pool[indexHolder][k5] + " , ");
 }
 System.out.println("\n");
}

private void initialize_chromosome_pool(int PopSize, int IntNumContracts)
{
 int i;
 int j;
 for (i = 0; i < PopSize; i++)
 {
  for (j = 0; j < IntNumContracts + 1; j++)
  {
   Pool[i][j] = Chromosome[i][j];
  }
 }
}

private void OnePoint_crossover_operator()
{
 double randNum;
 int cutPoint;
 int[] OffSpringOne = new int[IntNumContracts];
 int[] OffSpringTwo = new int[IntNumContracts];
 int[] ParentOne = new int[IntNumContracts];
 int[] ParentTwo = new int[IntNumContracts];
 int i;
 int j;
 int x;
 int z;
 int num1;
 int num2;
 Vector tabu = new Vector();
 for (int a = 0; a < PopSize; a++)
 {
  tabu.addElement(new Integer(a));
 }

 for (x = 0; x < PopSize / 2; x++)
 {
  randNum = Rand_Number.nextDouble();
```

```
if (randNum < OnePointCrossoverRate)
{
 randNum = Rand_Number.nextDouble();
 randNum = randNum * (IntNumContracts - 1) + 1;
 cutPoint = new Integer(Rounding.toString(randNum, 0)).intValue();
 cutPoint = cutPoint - 1;
 num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
    intValue();
 num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
    intValue();
 if (Pool[num1][0] > Pool[num2][0])
 {
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentOne[z - 1] = new Double(Pool[num2][z]).intValue();
  }
 }
 else
 {
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentOne[z - 1] = new Double(Pool[num1][z]).intValue();
  }
 }
 num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
    intValue();
 num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
    intValue();
 if (Pool[num1][0] > Pool[num2][0])
 {
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentTwo[z - 1] = new Double(Pool[num2][z]).intValue();
  }
 }
 else
 {
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentTwo[z - 1] = new Double(Pool[num1][z]).intValue();
  }
 }
 for (i = 0; i < cutPoint; i++)
 {
  OffSpringOne[i] = ParentOne[i];
  OffSpringTwo[i] = ParentTwo[i];
```

```
      }
      for (i = cutPoint; i < IntNumContracts; i++)
      {
       OffSpringOne[i] = ParentTwo[i];
       OffSpringTwo[i] = ParentOne[i];
      }
      //*****************
       for (i = 1; i < IntNumContracts + 1; i++)
       {
        Pool[counter][i] = OffSpringOne[i - 1];
        Pool[counter + 1][i] = OffSpringTwo[i - 1];
       }
      counter = counter + 2;
      NumOffspring = NumOffspring + 2;
    }
  }
}

private void Uniform_crossover_operator()
{
  int i;
  int j;
  int k;
  int x;
  int z;
  double randNum;
  int decisionVariable;
  int[] scrambleArray = new int[IntNumContracts];
  int[] OffSpringOne = new int[IntNumContracts];
  int[] OffSpringTwo = new int[IntNumContracts];
  int[] ParentOne = new int[IntNumContracts];
  int[] ParentTwo = new int[IntNumContracts];
  int decision;
  int num1;
  int num2;
  Vector tabu = new Vector();

  for (int a = 0; a < PopSize; a++)
  {
   tabu.addElement(new Integer(a));
  }
  for (x = 0; x < PopSize / 2; x++)
  {
   for (i = 0; i < IntNumContracts; i++)
   {
    randNum = Rand_Number.nextDouble();
```

```
  if (randNum < UniformCrossoverRate)
  {
   scrambleArray[i] = 1;
  }
  else
  {
   scrambleArray[i] = 0;
  }
}
num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
if (Pool[num1][0] > Pool[num2][0])
{
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentOne[z - 1] = new Double(Pool[num2][z]).intValue();
  }
}
else
{
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentOne[z - 1] = new Double(Pool[num1][z]).intValue();
  }
}
num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
   intValue();
if (Pool[num1][0] > Pool[num2][0])
{
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentTwo[z - 1] = new Double(Pool[num2][z]).intValue();
  }
}
else
{
  for (z = 1; z < IntNumContracts + 1; z++)
  {
   ParentTwo[z - 1] = new Double(Pool[num1][z]).intValue();
  }
}
for (k = 0; k < IntNumContracts; k++)
```

```
        {
         decision = scrambleArray[k];
         if (decision == 0)
         {
          OffSpringOne[k] = ParentOne[k];
          OffSpringTwo[k] = ParentTwo[k];
         }
         else if (decision == 1)
         {
          OffSpringOne[k] = ParentTwo[k];
          OffSpringTwo[k] = ParentOne[k];
         }
        }
        //*****************

        for (i = 1; i < IntNumContracts + 1; i++)
        {
         Pool[counter][i] = OffSpringOne[i - 1];
         Pool[counter + 1][i] = OffSpringTwo[i - 1];
        }
       counter = counter + 2;
       NumOffspring = NumOffspring + 2;
      }
     }

    private void mutation_operator()
    {
//Mutation #2: Local Seach with Tabu List
      int x;
      int z;
      int i;
      int j;
      int k;
      double randNum;
      int decisionVariable;
      int[] scrambleArray = new int[IntNumContracts];
      int[] OffSpring = new int[IntNumContracts];
      int[] Parent = new int[IntNumContracts];
      int decision;
      int num1;
      int num2;
      int P1 = 999;
      double tempz = 0;
      double tempzz;
      int number2;
      int number3;
```

```
      Integer number4;
      int listSize;
      int redo_flag = 0;

      Vector list = new Vector();
      Vector tabu = new Vector();

      for (int a = 0; a < PopSize; a++)
      {
       tabu.addElement(new Integer(a));
      }
      for (x = 0; x < PopSize; x++)
      {
        for (i = 0; i < IntNumContracts; i++)
       {
        randNum = Rand_Number.nextDouble();
        if (randNum < MutationRate2)
        {
         scrambleArray[i] = 1;
        }
        else
        {
         scrambleArray[i] = 0;
        }
       }

//_____________________________________________________________________

       randNum = Rand_Number.nextInt(tabu.size());
       int temp = new Double(randNum).intValue();
       num1 = ( (Integer) tabu.elementAt(temp)).intValue();

for (z = 1; z < IntNumContracts + 1; z++)
       {
        Parent[z - 1] = new Double(Pool[num1][z]).intValue();
        OffSpring[z - 1] = new Double(Pool[num1][z]).intValue();
       }
       number4 = new Integer(num1);
       list.addElement(number4);
       for (k = 0; k < IntNumContracts; k++)
       {
        decision = scrambleArray[k];
        if (decision == 0)
        {
         OffSpring[k] = Parent[k];
        }
```

```
else if (decision == 1)
{

  int size1;
  int Purchased = 0;
  double cumulativeTotal = 0;
  int index2;
  int period = 0;
  int index3;
  double Total;
  double tempy1;
  double tempy2;
  double tempy3;
  int turns;
  int numx;

  TCQ = new double[IntNumContracts];
  TCX = new double[IntNumContracts];
  for (j = 0; j < IntNumContracts; j++)
  {

    index2 = Contract_Period[j];
    if (j == k)
    {
      period = index2;
    }
    index3 = Contract_Length[j];
    size1 = OffSpring[j];
    TCQ[j] = rp_Array[index2 - 1] * ( (B[0] * index3) + B[2]);
    if (size1 > 0)
    {
      Purchased = 1;
    }
    else
    {
      Purchased = 0;
    }
    TCX[j] = rp_Array[index2 - 1] * B[1] * index3;

    Total = 0;
    Total = TCQ[j] * size1 + TCX[j] * Purchased;

    cumulativeTotal = cumulativeTotal + Total;
  }

  tempy1 = cumulativeTotal;
```

```
tempy3 = tempy1;
tempy2 = 0;
turns = 0;
while (tempy2 < tempy1)
{
 cumulativeTotal = 0;
 tempy1 = tempy3;
 turns = turns + 1;
 randNum = Rand_Number.nextDouble();
 randNum = randNum *
    ( (D[period - 1] * Rand_Number.nextDouble() * .3) - 1) + 1;
 numx = new Integer(Rounding.toString(randNum, 0)).intValue();
 numx = numx - 1;
 randNum = Rand_Number.nextDouble();
 if (Parent[k] > 0)
 {
  if (randNum <= .8)
  {
   tempz = Parent[k] + (numx * -1);
   if (tempz < 0)
   {
    tempz = 0;
   }
  }

  number2 = new Integer(Rounding.toString(tempz, 0)).intValue();
  OffSpring[k] = number2;
 }

 TCQ = new double[IntNumContracts];
 TCX = new double[IntNumContracts];

 for (j = 0; j < IntNumContracts; j++)
 {
  index2 = Contract_Period[j];
  index3 = Contract_Length[j];
  size1 = OffSpring[j];
  TCQ[j] = rp_Array[index2 -
     1] * ( (B[0] * index3) + B[2]);
  if (size1 > 0)
  {
   Purchased = 1;
  }
  else
  {
   Purchased = 0;
```

```
            }
            TCX[j] = rp_Array[index2 - 1] * B[1] * index3;

            Total = 0;
            Total = TCQ[j] * size1 + TCX[j] * Purchased;
            cumulativeTotal = cumulativeTotal + Total;
          }
          tempy2 = cumulativeTotal;
          tempy3 = tempy2;
          if (turns == 20)
          {
            OffSpring[k] = Parent[k];
            tempy1 = 0;
          }
        }
        turns = 0;
      }
    }
    //*****************

     for (i = 1; i < IntNumContracts + 1; i++)
     {
       Pool[counter][i] = OffSpring[i - 1];
     }
    counter = counter + 1;
    NumOffspring = NumOffspring + 1;
  }
}

private void mutation_operator_original()
{
  //Mutation #1: Tabu List with decreases in randomly selected genes *Not Used
  int x;
  int z;
  int i;
  int j;
  int k;
  double randNum;
  int decisionVariable;
  int[] scrambleArray = new int[IntNumContracts];
  int[] OffSpring = new int[IntNumContracts];
  int[] Parent = new int[IntNumContracts];
  int decision;
  int num1;
  int num2;
  int P1 = 999;
```

```
double tempz;
double tempzz;
double tempy;
int number2;
int number3;
Integer number4;
int listSize;
int redo_flag = 0;
int index2;
int period = 0;

Vector list = new Vector();
Vector tabu = new Vector();

for (int a = 0; a < PopSize; a++)
{
 tabu.addElement(new Integer(a));
}
for (x = 0; x < PopSize; x++)
{
 for (i = 0; i < IntNumContracts; i++)
 {
  randNum = Rand_Number.nextDouble();
  if (randNum < MutationRate1)
  {
   scrambleArray[i] = 1;
  }
  else
  {
   scrambleArray[i] = 0;
  }
 }
 //_____________________________________________________________________

 randNum = Rand_Number.nextInt(tabu.size());
 int temp = new Double(randNum).intValue();
 num1 = ( (Integer) tabu.elementAt(temp)).intValue();

 for (z = 1; z < IntNumContracts + 1; z++)
 {
  Parent[z - 1] = new Double(Pool[num1][z]).intValue();
  OffSpring[z - 1] = new Double(Pool[num1][z]).intValue();
 }
 number4 = new Integer(num1);
 list.addElement(number4);
 for (k = 0; k < IntNumContracts; k++)
```

```
    {
     decision = scrambleArray[k];
     if (decision == 0)
     {
      OffSpring[k] = Parent[k];
     }
     else if (decision == 1)
     {
      for (j = 0; j < IntNumContracts; j++)
      {
       index2 = Contract_Period[j];
       if (j == k)
       {
        period = index2;
       }
      }
      randNum = Rand_Number.nextDouble();
      tempz = randNum *
         ( (D[period - 1] * Rand_Number.nextDouble() * .3) - 1) + 1;
      number2 = new Integer(Rounding.toString(tempz, 0)).intValue();
      if (Parent[k] > 0)
      {
       if (Parent[k] > number2)
       {
        OffSpring[k] = Parent[k] - number2;
       }
       else
       {
        OffSpring[k] = 0;
       }
      }
     }
    }
    for (i = 1; i < IntNumContracts + 1; i++)
    {
     Pool[counter][i] = OffSpring[i - 1];
    }
    counter = counter + 1;
    NumOffspring = NumOffspring + 1;
   }
  }

  private void select_next_generation()
  {
   int i;
   int j;
```

```
  int x;
  int y;
  double randNum;
  int number;
  int num1;
  int num2;
  Vector tabu = new Vector();
  Chromosome[0] = this.bestChromosome.clone();
  for (int a = 0; a < NumOffspring + PopSize; a++)
  {
   tabu.addElement(new Integer(a));
  }
  for (i = 1; i < PopSize; i++)
  {
   randNum = Rand_Number.nextDouble();
   randNum = randNum * (NumOffspring + PopSize - 1) + 1;
   number = new Integer(Rounding.toString(randNum, 0)).intValue();
   number = number - 1;
   num1 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
      intValue();
   num2 = ( (Integer) tabu.elementAt(Rand_Number.nextInt(tabu.size()))).
      intValue();
   if (Pool[num1][0] < Pool[num2][0])
   {
    for (j = 0; j < IntNumContracts + 1; j++)
    {
     Chromosome[i][j] = Pool[num1][j];
    }
   }
   else
   {
    for (j = 0; j < IntNumContracts + 1; j++)
    {
     Chromosome[i][j] = Pool[num2][j];
    }
   }
  }
}

public void print_Chromosome_pool_with_FV()
{
  int start;
  int start1;
  System.out.println("Print Pool");
  for (start = 0; start < NumOffspring; start++)
  {
```

```
    for (start1 = 0; start1 < IntNumContracts; start1++)
    {
      System.out.print(Pool[start][start1] + " | ");
    }
    System.out.println("<-- PoolFitnessValue = " + Pool[start][0]);
  }
  System.out.println();
}

public void print_Chromosome_pool_withOUT_FV(int PopSize, int IntNumContracts)
{
  int start;
  int start1;
  System.out.println("Print Pool");
  for (start = 0; start < NumOffspring; start++)
  {
    for (start1 = 0; start1 < IntNumContracts; start1++)
    {
      System.out.print(Pool[start][start1] + " | ");
    }
    System.out.println();
  }
  System.out.println();
}

private void check_feasibility_pool(int NumPeriods)
{
  int k = 0;
  int i = 0;
  int j = 0;
  int temp1 = 0;
  int temp3 = 0;
  int offset;
  int index;
  int y;
  int[] Copy = new int[IntNumContracts];

  for (i = 1; i < IntNumContracts + 1; i++)
  {
    Copy[i - 1] = new Double(Pool[indexHolder][i]).intValue();
  }
  feasibilityFlagPool = 0;
  for (k = 1; k <= NumPeriods; k++)
  {
    offset = k - 1;
    index = offset;
```

```
    for (i = NumPeriods - k; i < NumPeriods; i++)
    {
     for (j = 1; j <= NumPeriods - k + 1; j++)
     {
      index = index + 1;
      temp1 = Copy[index - 1];
      temp3 = temp3 + temp1;
     }
     offset = offset - 1;
     index = index + offset;
    }
    if (D[k - 1] > temp3)
    {
     feasibilityFlagPool = 1;
    }
    temp3 = 0;
   }
}

private void repair_function_pool(int NumPeriods)
{
  NumContracts = .5 * NumPeriods * (NumPeriods + 1);
  IntNumContracts = new Double(NumContracts).intValue();

  int k = 0;
  int i = 0;
  int j = 0;
  int temp1 = 0;
  int temp3 = 0;
  int index = 0;
  double difference = 0;
  int sizeDifference = 0;
  double[] OffSpring1 = new double[IntNumContracts + 1];
  double[] OffSpring2 = new double[IntNumContracts + 1];
  double[][] DeficitArray = new double[NumPeriods][IntNumContracts + 1];
  int[] CommonContracts = new int[IntNumContracts + 1];
  int countPeriod3 = 0;
  int k5;
  int k6;
  int holder = 0;
  int holderB = 0;
  int maxNum;
  int ConCount;
  int maxNumIndex = 0;
  double deficitAmount;
  int turns = 0;
```

```
int numDeficit = IntNumContracts;
int requiredRepair = 0;
turns = 0;
numDeficit = NumPeriods + 1;
temp3 = 0;
index = 0;
temp1 = 0;
difference = 0;
sizeDifference = 0;
countPeriod3 = 0;
holder = 0;
ConCount = 0;
maxNum = 0;
maxNum = 0;
maxNumIndex = 0;
deficitAmount = 0;
holderB = 0;
int period = 0;
int offset_x = 0;
int[] Copy1 = new int[IntNumContracts + 1];
int[] Copy2 = new int[IntNumContracts + 1];
int[] Best2 = new int[IntNumContracts + 1];

for (i = 0; i < IntNumContracts + 1; i++)
{
 Copy1[i] = new Double(Pool[indexHolder][i]).intValue();
}
requiredRepair = 0;
while (numDeficit > 0)
{
 requiredRepair = 0;
 numDeficit = 0;
 turns = turns + 1;
 DeficitArray = new double[NumPeriods][IntNumContracts + 1];
 CommonContracts = new int[IntNumContracts + 1];

 for (i = 0; i < IntNumContracts; i++)
 {
  CommonContracts[i] = 0;
 }

 //--------------
 // This code finds the deficit amount for each period
 //--------------

 for (k = 1; k <= NumPeriods; k++)
```

```
        {
         temp3 = 0;
         if (k == 1)
         {
          index = k - 1;
          for (j = 1; j <= NumPeriods; j++)
          {
           index = index + 1;
           temp1 = 0;
           temp1 = Copy1[index];
           temp3 = temp3 + temp1;
          }
         }
         if (k > 1)
         {
          index = k - 1;
          offset_x = k - 2;
          for (i = 1; i <= k; i++)
          {
           for (j = k; j <= NumPeriods; j++)
           {
            index = index + 1;
            temp1 = 0;
            temp1 = Copy1[index];
            temp3 = temp3 + temp1;
           }
           index = index + offset_x;
           offset_x = offset_x - 1;
          }
         }
         if (temp3 < D[k - 1])
         {
          numDeficit = numDeficit + 1;
          requiredRepair = 1;
          difference = D[k - 1] - temp3;
          sizeDifference = new Double(Math.ceil(difference)).intValue();
          DeficitArray[k - 1][0] = sizeDifference;
         }
         else
         {
          DeficitArray[k - 1][0] = 0;
         }
        }

//-----------------------------
// This code fills in contracts for deficit periods
```

```
//------------------------

      for (k = 1; k <= NumPeriods; k++)
      {
       if (DeficitArray[k - 1][0] != 0)
       {
        if (k == 1)
        {
         countPeriod3 = 0;
         index = k - 1;

         //----------------------------------------
         // DeficitArray: One array for every period
         // first index in every array is the deficit for that period
         // Subsequent genes list the contracts which are active for a given period
         //----------------------------------------

         for (j = 1; j <= NumPeriods; j++)
         {
          index = index + 1;
          countPeriod3 = countPeriod3 + 1;
          DeficitArray[k - 1][countPeriod3] = index;
         }
        }
        if (k > 1)
        {
         countPeriod3 = 0;
         index = k - 1;
         offset_x = k - 2;
         for (i = 1; i <= k; i++)
         {

          //----------------------------------------
          // DeficitArray: One array for every period
          // first index in every array is the deficit for that period
          // Subsequent genes list the contracts which are active for a given period
          //----------------------------------------

          for (j = k; j <= NumPeriods; j++)
          {
           index = index + 1;
           countPeriod3 = countPeriod3 + 1;
           DeficitArray[k - 1][countPeriod3] = index;
          }
          index = index + offset_x;
          offset_x = offset_x - 1;
```

```
      }
    }
  }
}

//---------------------------------------------
// This code counts, for each contract, the number of times
//  it appears in a deficit period
//---------------------------------------------
int holder2 = 0;
for (k5 = 0; k5 < NumPeriods; k5++)
{
 if (DeficitArray[k5][0] != 0)
 {
  for (k6 = 1; k6 < IntNumContracts + 1; k6++)
  {
   holder = new Double(DeficitArray[k5][k6]).intValue();
   if (holder != 0)
   {
    ConCount = CommonContracts[holder];
    ConCount = ConCount + 1;
    CommonContracts[holder] = ConCount;
   }
  }
 }
}

//---------------------------------------------
// NumCommonContracts = variable containing the total number of
// contracts in periods with deficits
//---------------------------------------------

CommonContracts[0] = 0;

//-----------------------
// The code that follows assigns the index number of the 1st contract to
// have the highest number of occurances in a deficit period to variable
// maxNumIndex; the highest number of occurances is stored in maxNum
//---------------------------------------

maxNum = 0;
maxNumIndex = 0;
for (k6 = 1; k6 <= IntNumContracts; k6++)
{
 holder = CommonContracts[k6];
 if (holder >= maxNum)
```

```
    {
     maxNum = holder;
     maxNumIndex = k6;
    }
   }
  if (maxNumIndex > 0)
  {
   for (k5 = 0; k5 < NumPeriods; k5++)
   {
    if (DeficitArray[k5][0] != 0)
    {
     for (k6 = 1; k6 < IntNumContracts + 1; k6++)
     {
      if (maxNumIndex == DeficitArray[k5][k6])
      {
       period = k5;
       k6 = IntNumContracts + 1;
       k5 = NumPeriods;
      }
     }
    }
   }

   deficitAmount = new Double(DeficitArray[period][0]).intValue();
   DeficitArray[period][0] = 0;

   //----------------------
   // holderB = the deficitAmount for a given period divided by
   // the number of contracts of the given period
   //-----------------------

   holderB = new Integer(Rounding.toString(deficitAmount, 0)).intValue();
   holder = Copy1[maxNumIndex];
   holder = holder + holderB;
   Copy1[maxNumIndex] = holder;
  }
}

double Total;
double cumulativeTotal;
int index2;
int index3;
double size1;

Total = 0;
cumulativeTotal = 0;
```

```
for (j = 1; j <= IntNumContracts; j++)
{
 index2 = Contract_Period[j - 1];
 index3 = Contract_Length[j - 1];
 size1 = Copy1[j];
 if (size1 > 0)
 {
  /*********************************/
  TCQ[j - 1] = rp_Array[index2 - 1] * ( (B[0] * index3) + B[2]);
  TCX[j - 1] = rp_Array[index2 - 1] * B[1] * index3;
  Total = 0;
  Total = TCQ[j - 1] * size1 + TCX[j - 1];
  /*******************************/

  cumulativeTotal = cumulativeTotal + Total;
  Copy1[0] = new Double(cumulativeTotal).intValue();
 }
}

for (i = 0; i < IntNumContracts + 1; i++)
{
 OffSpring1[i] = new Integer(Copy1[i]).doubleValue();
}
temp1 = 0;
temp3 = 0;
index = 0;
difference = 0;
sizeDifference = 0;
feasibilityFlagPool = 0;

Total = 0;
cumulativeTotal = 0;

index3 = 0;
for (i = 0; i < IntNumContracts + 1; i++)
{
 CommonContracts[i] = 0;
 for (j = 0; j < NumPeriods; j++)
 {
  DeficitArray[j][i] = 0;
 }
}

period = 0;
turns = turns + 1;
temp3=0;
```

```
for (i = 0; i < IntNumContracts + 1; i++)
{
 Copy2[i] = new Double(Pool[indexHolder][i]).intValue();
 Best2[i] = new Double(Pool[indexHolder][i]).intValue();
}
Best2[0]=9999999;
for(int alpha=0;alpha<20;alpha++)
{
 for (i = 0; i < NumPeriods; i++)
 {
  for (j = 0; j <= IntNumContracts; j++)
  {
   DeficitArray[i][j] = 0;
   CommonContracts[j] = 0;
  }
 }
 numDeficit = 1;
 while (numDeficit > 0)
 {
  requiredRepair = 0;
  numDeficit = 0;
  turns = turns + 1;
  DeficitArray = new double[NumPeriods][IntNumContracts + 1];
  //--------------
  // This code finds the deficit amount for each period
  //--------------

  for (k = 1; k <= NumPeriods; k++)
  {
   temp3 = 0;
   if (k == 1)
   {
    index = k - 1;
    for (j = 1; j <= NumPeriods; j++)
    {
     index = index + 1;
     temp1 = 0;
     temp1 = Copy2[index];
     temp3 = temp3 + temp1;
    }
   }
   if (k > 1)
   {
    index = k - 1;
    offset_x = k - 2;
```

```
            for (i = 1; i <= k; i++)
            {
             for (j = k; j <= NumPeriods; j++)
             {
              index = index + 1;
              temp1 = 0;
              temp1 = Copy2[index];
              temp3 = temp3 + temp1;
             }
             index = index + offset_x;
             offset_x = offset_x - 1;
            }
           }
           if (temp3 < D[k - 1])
           {
            numDeficit = numDeficit + 1;
            requiredRepair = 1;
            difference = D[k - 1] - temp3;
            sizeDifference = new Double(Math.ceil(difference)).intValue();
            DeficitArray[k - 1][0] = sizeDifference;
           }
           else
           {
            DeficitArray[k - 1][0] = 0;
           }
          }
          if (numDeficit == 0)
           break;
//----------------------------
// This code fills in contracts for deficit periods
//------------------------

          for (k = 1; k <= NumPeriods; k++)
          {
           if (DeficitArray[k - 1][0] != 0)
           {
            if (k == 1)
            {
             countPeriod3 = 0;
             index = k - 1;

             //----------------------------------------
             // DeficitArray: One array for every period
             // first index in every array is the deficit for that period
             // Subsequent genes list the contracts which are active for a given period
             //----------------------------------------
```

```
      for (j = 1; j <= NumPeriods; j++)
      {
       index = index + 1;
       countPeriod3 = countPeriod3 + 1;
       DeficitArray[k - 1][countPeriod3] = index;
      }
     }
     if (k > 1)
     {
      countPeriod3 = 0;
      index = k - 1;
      offset_x = k - 2;
      for (i = 1; i <= k; i++)
      {

       //----------------------------------------
       // DeficitArray: One array for every period
       // first index in every array is the deficit for that period
       // Subsequent genes list the contracts which are active for a given period
       //----------------------------------------

       for (j = k; j <= NumPeriods; j++)
       {
        index = index + 1;
        countPeriod3 = countPeriod3 + 1;
        DeficitArray[k - 1][countPeriod3] = index;
       }
       index = index + offset_x;
       offset_x = offset_x - 1;
      }
     }
    }
}

for (int l = NumPeriods - 1; l >= 0; l--)
{
 if (DeficitArray[l][0] != 0)
 {
  period = l;
  break;
 }
}
int counter = 0;
for (int l = 1; l < DeficitArray[period].length; l++)
{
```

```
   if (DeficitArray[period][l] == 0)
   {
    counter = l - 1;
    break;
   }
  }
  int rand =1+ this.Rand_Number.nextInt(counter-1);

  int contract = new Double(DeficitArray[period][rand]).intValue();
  deficitAmount = new Double(DeficitArray[period][0]).intValue();
  DeficitArray[period][0] = 0;

  //----------------------
  // holderB = the deficitAmount for a given period divided by
  // the number of contracts of the given period
  //-----------------------

  holderB = new Integer(Rounding.toString(deficitAmount, 0)).intValue();
  holder = Copy2[contract];
  holder = holder + holderB;
  Copy2[contract] = holder;
}

Total = 0;
cumulativeTotal = 0;
for (j = 1; j <= IntNumContracts; j++)
{
  index2 = Contract_Period[j - 1];
  index3 = Contract_Length[j - 1];
  size1 = Copy2[j];
  if (size1 > 0)
  {
   /*********************************/
   TCQ[j - 1] = rp_Array[index2 - 1] * ( (B[0] * index3) + B[2]);
   TCX[j - 1] = rp_Array[index2 - 1] * B[1] * index3;
   Total = 0;
   Total = TCQ[j - 1] * size1 + TCX[j - 1];
   /*********************************/

   cumulativeTotal = cumulativeTotal + Total;
   Copy2[0] = new Double(cumulativeTotal).intValue();
  }
}
if(Best2[0]>Copy2[0])
{
  for(int beta=0;beta<this.IntNumContracts;beta++)
```

```
    {

      Best2[beta] = Copy2[beta];
    }
   }
   for(int m=0;m<this.IntNumContracts;m++)
   {
    Copy2[m]= new Double(Pool[indexHolder][m]).intValue();
   }
}
for(int beta=0;beta<this.IntNumContracts;beta++)
{
 Copy2[beta] = Best2[beta];
}

for (i = 0; i < IntNumContracts + 1; i++)
{
 OffSpring2[i] = new Integer(Copy2[i]).doubleValue();
}

temp1 = 0;
temp3 = 0;
index = 0;
difference = 0;
sizeDifference = 0;

if (OffSpring1[0] < OffSpring2[0])
{
 for (i = 0; i < IntNumContracts + 1; i++)
 {
  Pool[indexHolder][i] = OffSpring1[i];
 }
 Repair1 = Repair1 + 1;
}

if (OffSpring1[0] == OffSpring2[0])
{
 for (i = 0; i < IntNumContracts + 1; i++)
 {
  Pool[indexHolder][i] = OffSpring1[i];
 }
 RepairTie = RepairTie + 1;
}

if (OffSpring1[0] > OffSpring2[0])
{
```

```
    for (i = 0; i < IntNumContracts + 1; i++)
    {
     Pool[indexHolder][i] = OffSpring2[i];
    }
    Repair2 = Repair2 + 1;
   }
  }

 private int record_best_chromosome_from_Pool(int NumPeriods)
 {
  int i;
  int j;
  int x;
  int y;
  int count = 0;
  Vector Best = new Vector();
  Vector Test = new Vector();
  int same = 0;

  NumContracts = .5 * NumPeriods * (NumPeriods + 1);
  IntNumContracts = new Double(NumContracts).intValue();
  change = 0;

  if (gen == 0)
  {
   for (i = 0; i < NumOffspring + this.PopSize; i++)
   {
    if (i == 0)
    {
     for (j = 0; j < IntNumContracts + 1; j++)
     {
      bestChromosome[j] = Pool[i][j];
     }
    }
    if (i > 0)
    {
     if (bestChromosome[0] > Pool[i][0])
     {
      for (j = 0; j < IntNumContracts + 1; j++)
      {
       bestChromosome[j] = Pool[i][j];
      }
     }
     else if (bestChromosome[0] == Pool[i][0])
     {
      for (j = 0; j < IntNumContracts + 1; j++)
```

```
        {
         if (bestChromosome[j] != Pool[i][j])
         {
          same = 1;
          change = 1;
         }
        }
        if (same == 1)
        {
         for (j = 0; j < IntNumContracts + 1; j++)
         {
          bestChromosome[j] = Pool[i][j];
         }
        }
       }
      }
     }
    }
    else if (gen > 0)
    {
     for (i = 0; i < NumOffspring; i++)
     {
      if (bestChromosome[0] > Pool[i][0])
      {
       change = 1;
       for (j = 0; j < IntNumContracts + 1; j++)
       {
        bestChromosome[j] = Pool[i][j];
       }
      }
      else if (bestChromosome[0] == Pool[i][0])
      {
       for (j = 0; j < IntNumContracts + 1; j++)
       {
        if (bestChromosome[j] != Pool[i][j])
        {
         same = 1;
         change = 1;
        }
       }
       if (same == 1)
       {
        for (j = 0; j < IntNumContracts + 1; j++)
        {
         bestChromosome[j] = Pool[i][j];
        }
```

```
        }
       }
     }
   }
   return change;
}

public void write_to_file(boolean first_time, String FileName)
{
   int p = new Integer(Rounding.toString(gp * 10, 0)).intValue();
   int ad_d = new Integer(Rounding.toString(ad * 1, 0)).intValue();
   int d = new Integer(Rounding.toString(gd * 10, 0)).intValue();
   if (FileName.equals(""))
   {
     this.write = new WriteFile(buff, FileNum);
     if (first_time == true)
     {
       this.write.write_toFile();
     }
     else
     {
       this.write.append_toFile();
     }
     this.buff.delete(0, this.buff.length());
     this.buff = new StringBuffer();
   }
   else
   {
     this.write = new WriteFile(buff2, FileNum, FileName);
     if (first_time == true)
     {
       this.write.write_toFile();
     }
     else
     {
       this.write.append_toFile();
     }
     this.buff2.delete(0, this.buff2.length());
     this.buff2 = new StringBuffer();
   }
}

public void Fill_Best_Vector()
{
   int i;
   int j;
```

```
int k;
int Size;
double[] tempA;
double[] tempB;
NumContracts = .5 * NumPeriods * (NumPeriods + 1);
IntNumContracts = new Double(NumContracts).intValue();

tempA = new double[IntNumContracts + 1];
tempB = new double[IntNumContracts + 1];
int Size_of_Vector = 20;

if (gen == 0)
{
 for (k = 0; k < Size_of_Vector; k++)
 {
  tempB[0] = 999999999;
  Fill_Best.addElement(tempB);
 }
}
for (i = 0; i < (NumOffspring + PopSize); i++)
{
 for (j = Size_of_Vector - 1; j >= 0; j--)
 {
  tempA = (double[]) Fill_Best.elementAt(j);
  if (Pool[i][0] < tempA[0])
  {
   if (j != 0)
   {
    continue;
   }
   else
   {
    Fill_Best.insertElementAt( (double[]) Pool[i].clone(), 0);
    Fill_Best.removeElementAt(Fill_Best.size() - 1);
   }
  }
  else if (Pool[i][0] == tempA[0])
  {
   tempB = (double[]) Pool[i].clone();
   int p;
   for (p = 0; p < IntNumContracts + 1; p++)
   {
    if (tempA[p] != tempB[p])
    {
     p = IntNumContracts + 2;
     Fill_Best.insertElementAt( (double[]) Pool[i].clone(), j + 1);
```

```
            Size = Fill_Best.size();
            if (Size > Size_of_Vector)
            {
              Fill_Best.removeElementAt(Fill_Best.size() - 1);
            }
          }
        }
        break;
      }
      else if (Pool[i][0] > tempA[0])
      {
        if (Size_of_Vector - 1 == j)
        {
          Fill_Best.addElement( (double[]) Pool[i].clone());
        }
        else
        {
          Fill_Best.insertElementAt( (double[]) Pool[i].clone(), j + 1);
        }
        Size = Fill_Best.size();
        if (Size > Size_of_Vector)
        {
          Fill_Best.removeElementAt(Fill_Best.size() - 1);
        }
        break;
      }
    }
  }
  Fill_Best.trimToSize();
}

//If you send the timeperiod it will return the index number of all
// valid contracts in a Vector
public Vector valid_contracts(int timeperiod)
{
    Vector valid = new Vector();
    int x = this.NumPeriods;
    int y = timeperiod;
    for (int i =1;i<=timeperiod;i++)
    {
        for(int j=x;j>=y;j--)
        {
            valid.addElement(new Integer(j));
        }
        x= x+this.NumPeriods-i;
        y=y+this.NumPeriods -i;
```

```
        }
        valid.trimToSize();
        return valid;
      }

/**********************************************************************/
/*******************START SIMULATION FUNCTIONS*********************/
/**********************************************************************/
/*Updating demand array with average demand from simulation*******************/
      public void sim_update_demand(double[][] demand)
      {
        double value = 0;
        for(int i=0;i<this.NumPeriods;i++)
        {
           int x = demand[i].length;
          for(int j=0;j<demand[i].length;j++)
          {
            value = value + demand[i][j];
          }
          this.D[i] = value/demand[i].length;
          value=0;
        }
      }
/*Updating price array with average price from simulation********************/
      public void sim_update_price(double[][] price)
      {
        double value = 0;
        for(int i=0;i<this.NumPeriods;i++)
        {
          for(int j=0;j<price[i].length;j++)
          {
            value = value + price[i][j];
          }
          this.rp_Array[i] = value/price[i].length;
          value=0;
        }
      }

      public double find_deficit(double[] chromosome, int Period, int instance)
      {
        double deficit = 0;
        double contract_amount = 0;
        Vector temp = this.valid_contracts(Period+1);
        for(int i =0;i<temp.size();i++)
        {
          int index = ((Integer)temp.elementAt(i)).intValue();
```

```
        contract_amount = contract_amount+chromosome[index];
      }
      if(instance >=0)
      {
       deficit = this.sim_demand[Period][instance] - contract_amount;
      }
      else
      {
       deficit = this.D[Period] - contract_amount;
      }
      if(deficit<0)
       deficit = 0;
      return deficit;
     }

    public double[] sim_evaluate_chromosome(double[] chromosome, int
chromosome_index)
     {
      this.NumContracts = .5 * this.NumPeriods * (this.NumPeriods + 1);
      this.IntNumContracts = new Double(NumContracts).intValue();
      double [] sim_evaluation = new double[this.sim_instances];
      double avgTotal = 0;
      for(int i=0;i<this.NumPeriods;i++)
      {
       for(int j=0;j<this.sim_instances;j++)
       {
        double deficit = this.find_deficit(chromosome,i,j);
        if (deficit !=0)
        {
         sim_evaluation[j]=sim_evaluation[j]+deficit*(this.sim_price[i][j]*
             B[0]*1+B[2])+this.sim_price[i][j]*
             B[1]*1;
        }
       }
      }
      for (int i=0;i<this.IntNumContracts;i++)
      {
       int contract_period = this.Contract_Period[i];
       int contract_length = this.Contract_Length[i];
       double contract_amount = chromosome[i+1];
       for(int j =0;j<this.sim_instances;j++)
       {
        if(contract_amount != 0)
        {
```

```
sim_evaluation[j]=sim_evaluation[j]+contract_amount*(this.sim_price[contract_period-
1][j]*
                B[0]*contract_length+B[2])+this.sim_price[contract_period-1][j]*
                B[1]*contract_length;
          }
         }
        }
        this.sim_chromosome_value[chromosome_index]= sim_evaluation.clone();
        for(int i =0;i<this.sim_instances;i++)
        {
         avgTotal = avgTotal + sim_evaluation[i];
        }
        avgTotal = avgTotal/this.sim_instances;
        chromosome[0]= avgTotal;
        return chromosome;
       }

       public double [] training_results(double[] chromosome, int chromosome_index)
       {
        this.NumContracts = .5 * this.NumPeriods * (this.NumPeriods + 1);
        this.IntNumContracts = new Double(NumContracts).intValue();
        double [] sim_evaluation = new double[this.sim_instances];
        double avgTotal = 0;
        for(int i=0;i<this.NumPeriods;i++)
        {
         for(int j=0;j<this.sim_instances;j++)
         {
          double deficit = this.find_deficit(chromosome,i,j);
          if (deficit !=0)
          {
           sim_evaluation[j]=sim_evaluation[j]+deficit*this.sim_price[i][j]*
              (B[0]*1+B[2])+this.sim_price[i][j]*
              B[1]*1;
          }
         }
        }
        for (int i=0;i<this.IntNumContracts;i++)
        {
         int contract_period = this.Contract_Period[i];
         int contract_length = this.Contract_Length[i];
         double contract_amount = chromosome[i+1];
         for(int j =0;j<this.sim_instances;j++)
         {
          if(contract_amount != 0)
          {
```

```
sim_evaluation[j]=sim_evaluation[j]+contract_amount*this.sim_price[contract_period-
1][j]*
                (B[0]*contract_length+B[2])+this.sim_price[contract_period-1][j]*
                B[1]*contract_length;
            }
           }
          }
          return sim_evaluation;
        }

/*************************************************************************/
/************************END SIMULATION FUNCTIONS*************************/
/*************************************************************************/

    /*************** MAIN ***************/
    public static void main(String[] args)
    {
     start ea1 = new start();
     System.out.println("Begin EA");
     ea1.NumPeriods = 6;
     ea1.PopSize = 40;
     ea1.maxGenerations = 500;
     int n = 0;
     int z;
     int ProblemNumber = 0;
     int x;
     int y;
     int count = 0;
     int term = 0;
     double Term_Condition;
     double prob_purchase;
     double randNum;
     int num1;
     int j, k;
     double number;
     int number2;
     int number3;
     double tempz;
     double tempzz;
     double tempy;
     double Restart = 0;
     int Restart_Counter = 0;
     int Gen_Best_Found = 0;
     long TimetoComplete = 0;
     long TimetoBest = 0;
```

```
int Size = 1;
double increment_gp = .02;
double increment_gd = .02;
int increment_n = 6;
int increment_Size = 1;
int x_count = 0;
int y_count = 0;
int z_count = 0;
String econScale = "x";
ea1.n_scale = 0;
int tempRepair1 = 0;
int tempRepair2 = 0;
int tempRepairTie = 0;
ea1.gp = .05;
ea1.ad = 540;
ea1.gd = .05;
ea1.Date = 10.26;
int i;
int runNumber;
int Bingo = 0;
int numRuns;
int remainder;
int numProb;
int AdjPopSize;
int AdjMaxGen;
int AdjInitialDemand;
int incrementPrice = 0;

ea1.UniformCrossoverRate = .8;
ea1.OnePointCrossoverRate = .2;
ea1.MutationRate2 = .6;
ea1.MutationRate1 = .6;

ea1.NumPeriods = 0;
for (int x_n = 1; x_n <= 3; x_n
{
 int buff_Length = ea1.buff.length();
 ea1.buff.delete(0, buff_Length);
 ea1.buff = new StringBuffer();
 if (x_n==1)
 {
  ea1.NumPeriods=6;
 }
 else if (x_n==2)
 {
  ea1.NumPeriods=12;
```

```
}
else if(x_n==3)
{
 ea1.NumPeriods=18;
}
else
{
 System.exit(0);
}
ea1.n_scale = ea1.n_scale + 1;
ea1.NumContracts = .5 * ea1.NumPeriods * (ea1.NumPeriods + 1);
ea1.IntNumContracts = new Double(ea1.NumContracts).intValue();
ea1.PoolSize = ea1.PopSize * 200;
ea1.Best_All_Runs = new double[ea1.IntNumContracts + 1];
ea1.bestChromosome = new double[ea1.IntNumContracts + 1];
double[] overall_Best = new double[ea1.IntNumContracts + 1];
int[] OffSpring1 = new int[ea1.IntNumContracts];
int[] OffSpring2 = new int[ea1.IntNumContracts];
Size = 0;
y_count = 0;

for (int x_Size = 1; x_Size <= 3; x_Size++)
{
 ea1.gen = 0;
 Size = Size + 1;
 y_count = 0;
 ea1.gp = .05;
 ea1.gd = .05;

 buff_Length = ea1.buff.length();
 ea1.buff.delete(0, buff_Length);
 ea1.buff = new StringBuffer();

 if (Size == 1)
 {
  //Small economies of scale
  /* */
  ea1.B1 = 13.116;
  ea1.B2 = 962.927;
  ea1.B3 = 258.526;
  econScale = "econScale-Small";
  /* */
 }
 if (Size == 2)
 {
  //Medium economies of scale
```

```
          /* */
          ea1.B1 = 6.757;
          ea1.B2 = 928.503;
          ea1.B3 = 364.989;
          econScale = "econScale-Medium";
          /* */
        }
        if (Size == 3)
        {
          //Large economies of scale
          /* */
          ea1.B1 = 0.3977;
          ea1.B2 = 894.115;
          ea1.B3 = 471.416;
          econScale = "econScale-Large";
          /* */
        }

        ea1.B[0] = ea1.B1;
        ea1.B[1] = ea1.B2;
        ea1.B[2] = ea1.B3;

        for (numProb = 1; numProb <= 9; numProb++)
        {
          ProblemNumber = ProblemNumber + 1;
          y_count = y_count + 1;
          ea1.gen = 0;
          ea1.NumContracts = .5 * ea1.NumPeriods * (ea1.NumPeriods + 1);
          ea1.IntNumContracts = new Double(ea1.NumContracts).intValue();
          ea1.fill_arrays();
          ea1.FileNum = ea1.FileNum + 1;
          runNumber = 0;
          int MaxRuns = 1;
          double overall_BestofRuns = 0;
          double TimetoBestofRuns = 0;
          double TimetoCompleteforAllRuns = 0;
          int Gen_BestofRuns = 0;
          double[][] runFitnessTimes = new double[MaxRuns][5];
          double TotalTimetoCompleteforAllRuns = 0;
/*************************************************************************/
/************************SIMULATION CODE**********************************/
/*************************************************************************/
          ea1.sim1 = new sim(ea1.NumPeriods,ea1.B, false, false,ProblemNumber);
          int distribution = 1;//1 for Uniform, 2 for Normal, 3 for Exponential
          double d_spread = 0.1;// This gives the width for Uniform and std dev for Normal
distribution (< 1)
```

```
        double p_spread = 0.1;

        if(ea1.gd == 0.05)
        {
         d_spread = 0.1;
        }
        else if (ea1.gd == 0.07)
        {
         d_spread = 0.06;
        }
        else if (ea1.gd == 0.09)
        {
         d_spread = 0.02;
        }
        if(ea1.gp == 0.05)
        {
         p_spread = 0.1;
        }
        else if (ea1.gp == 0.07)
        {
         p_spread = 0.06;
        }
        else if (ea1.gp == 0.09)
        {
         p_spread = 0.02;
        }
        ea1.sim1.populate(distribution,ea1.gp,ea1.gd,d_spread, p_spread);
        ea1.sim_price = ea1.sim1.price.clone();
        ea1.sim_demand = ea1.sim1.demand.clone();
        ea1.sim_update_demand(ea1.sim_demand);
        ea1.sim_update_price(ea1.sim_price);
        ea1.sim_instances = ea1.sim1.NumSim;
/*****************************************************************/
        System.out.println("Solving Problem Number " + ProblemNumber);
        System.out.println("N = " + ea1.NumPeriods + ", gp = " + ea1.gp +
                        ", ad = " + ea1.ad + ", d_spread = "+ d_spread +
                        ", p_spread = "+ p_spread +
                        ", gd = " + ea1.gd + ", B1 = " + ea1.B1 +
                        ", B2 = " + ea1.B2 + ", B3 = " + ea1.B3 +
                        ", # best the same last value = " + count +
                        ", PopSize = " + ea1.PopSize +
                        ", MaxGeneration = " + ea1.maxGenerations);
        System.out.println(econScale + ", UniformCrossoverRate = " +
                        ea1.UniformCrossoverRate +
                        ", OnePointCrossoverRate = " +
                        ea1.OnePointCrossoverRate +
```

```
                ", MutationRate2 = " + ea1.MutationRate2 +
                ", ea1.MutationRate1 = " + ea1.MutationRate1);
System.out.println("PopSize = " + ea1.PopSize +
                ", MaxGeneration = " + ea1.maxGenerations +
                ", # Restarts = " + Restart_Counter);
ea1.temp = "N = "+ea1.NumPeriods+", gp = "+ea1.gp+", ad = "+ea1.ad
   +", gd = "+ea1.gd+", d_spread = "+ d_spread+", p_spread = "+
   p_spread +", B1 = "+ea1.B1+" , B2 = "+ea1.B2+", B3 = "
   +ea1.B3+", PopSize = "+ea1.PopSize+", MaxGeneration = "+
   ea1.maxGenerations;
ea1.buff.append(ea1.temp + "\r\n");
ea1.temp = econScale + ", UniformCrossoverRate = " +
   ea1.UniformCrossoverRate +
   ", OnePointCrossoverRate = " + ea1.OnePointCrossoverRate +
   ", MutationRate2 = " + ea1.MutationRate2 +
   ", ea1.MutationRate1 = " + ea1.MutationRate1;
ea1.buff.append(ea1.temp + "\r\n");
ea1.temp = "PopSize = " + ea1.PopSize + ", MaxGeneration = " +
   ea1.maxGenerations;
ea1.buff.append(ea1.temp + "\r\n");

for (numRuns = 1; numRuns <= MaxRuns; numRuns++)
{
 ea1.gen = 0;
 ea1.initialize_chromosomes();
 long Start = System.currentTimeMillis();
 TimetoComplete = 0;
 TimetoBest = 0;
 term = 0;
 Restart = 0;
 Restart_Counter = 0;
 overall_Best[0] = 999999999;
 Gen_Best_Found = 0;
 count = 0;
 runNumber = runNumber + 1;
 System.out.println(" ");
 ea1.buff.append("\n");
 System.out.println("Problem Number = " + ProblemNumber +
                ", Run Number = " + runNumber);
 ea1.temp = "Problem Number = " + ProblemNumber +
    ", Run Number = " + runNumber;
 ea1.buff.append(ea1.temp + "\r\n");
 ea1.counter = ea1.PopSize;
 for (ea1.gen = 0; ea1.gen <= ea1.maxGenerations; ea1.gen++)
 {
  ea1.NumOffspring = ea1.PopSize;
```

```
            ea1.counter = ea1.PopSize;
            ea1.initialize_chromosome_pool(ea1.PopSize, ea1.IntNumContracts);
            ea1.mutation_operator();
            ea1.OnePoint_crossover_operator();
            ea1.Uniform_crossover_operator();
            ea1.Repair1 = 0;
            ea1.Repair2 = 0;
            ea1.RepairTie = 0;
            for (ea1.indexHolder = 0;
                ea1.indexHolder < (ea1.NumOffspring + ea1.PopSize);
                ea1.indexHolder++)
            {
             ea1.repair_function_pool(ea1.NumPeriods);
             tempRepair1 = tempRepair1 + ea1.Repair1;
             tempRepair2 = tempRepair2 + ea1.Repair2;
             tempRepairTie = tempRepairTie + ea1.RepairTie;
            }
/*********************************************************************/
/*****************SIM CODING**********************************************/
/*********************************************************************/
            ea1.sim_chromosome_value = new double[ea1.NumOffspring +
ea1.PopSize][ea1.sim_instances];
            for(i =0;i<(ea1.NumOffspring + ea1.PopSize);i++)
            {
             double[] chromosome = ea1.sim_evaluate_chromosome(ea1.Pool[i],i);
             ea1.Pool[i] = chromosome.clone();
            }
/*********************************************************************/

            ea1.change = ea1.record_best_chromosome_from_Pool(ea1.NumPeriods);
            ea1.Fill_Best_Vector();
            if (ea1.change == 0)
            {
             count = count + 1;
            }
            else if (ea1.change == 1)
            {
             count = 0;
            }
            term = ea1.gen;
            if (overall_Best[0] > ea1.bestChromosome[0])
            {
             Gen_Best_Found = ea1.gen;
             for (i = 0; i < ea1.IntNumContracts + 1; i++)
             {
              overall_Best[i] = ea1.bestChromosome[i];
```

```
    }
    long ToBest = System.currentTimeMillis();
    TimetoBest = ToBest - Start;
  }
  for (i = 0; i < ea1.PopSize; i++)
  {
    for (j = 0; j < ea1.IntNumContracts + 1; j++)
    {
      ea1.Chromosome[i][j] = 0;
    }
  }
  for (ea1.indexHolder = 0;
       ea1.indexHolder < ea1.IntNumContracts + 1;
       ea1.indexHolder++)
  {
    ea1.Chromosome[0][ea1.indexHolder] =
        ea1.bestChromosome[ea1.indexHolder];
  }

  ea1.select_next_generation();
  for (i = 0; i < ea1.PoolSize; i++)
  {
    for (j = 0; j < ea1.IntNumContracts + 1; j++)
    {
      ea1.Pool[i][j] = 0;
    }
  }
  if (count >= 100 * ea1.NumPeriods)
  {
    term = ea1.gen;
    ea1.gen = ea1.maxGenerations;
  }
}

long End = System.currentTimeMillis();
TimetoComplete = End - Start;
runFitnessTimes[numRuns - 1][0] = overall_Best[0];
runFitnessTimes[numRuns - 1][1] = Gen_Best_Found;
runFitnessTimes[numRuns - 1][2] = TimetoBest;
runFitnessTimes[numRuns - 1][3] = TimetoComplete;

if (numRuns == 1)
{
  overall_BestofRuns = runFitnessTimes[numRuns - 1][0];
  Gen_BestofRuns = new Double(runFitnessTimes[numRuns - 1][1]).
      intValue();
```

```
  TotalTimetoCompleteforAllRuns = TimetoComplete;
  TimetoBestofRuns = runFitnessTimes[numRuns - 1][2];

  for (z = 0; z < ea1.IntNumContracts + 1; z++)
  {
   ea1.Best_All_Runs[z] = ea1.bestChromosome[z];
  }
}
else
{
 if (runFitnessTimes[numRuns - 2][0] >
    runFitnessTimes[numRuns - 1][0])
 {
  overall_BestofRuns = runFitnessTimes[numRuns - 1][0];
  Gen_BestofRuns = new Double(runFitnessTimes[numRuns - 1][1]).
     intValue();
  TimetoBestofRuns = runFitnessTimes[numRuns - 1][2];
  for (j = 0; j < ea1.IntNumContracts + 1; j++)
  {
   ea1.Best_All_Runs[j] = ea1.bestChromosome[j];
  }
 }
 else
 {
  if (runFitnessTimes[numRuns - 2][0] ==
     runFitnessTimes[numRuns - 1][0] &&
     runFitnessTimes[numRuns - 2][1] >
     runFitnessTimes[numRuns - 1][1])
  {
   overall_BestofRuns = runFitnessTimes[numRuns - 1][0];
   Gen_BestofRuns = new Double(runFitnessTimes[numRuns - 1][1]).
      intValue();
   TimetoBestofRuns = runFitnessTimes[numRuns - 1][2];
   for (j = 0; j < ea1.IntNumContracts + 1; j++)
   {
    ea1.Best_All_Runs[j] = ea1.bestChromosome[j];
   }
  }
 }
 TotalTimetoCompleteforAllRuns = TotalTimetoCompleteforAllRuns +
    TimetoComplete;
}
System.out.println("Gen # Run terminated = " + term);
ea1.temp = "Gen # Run terminated = " + term;
ea1.buff.append(ea1.temp + "\r\n");
System.out.println("Gen to find Best Chromosome this run = " +
```

```
                    runFitnessTimes[numRuns - 1][1]);
            ea1.temp = "Gen to find Best Chromosome this run = " +
               runFitnessTimes[numRuns - 1][1];
            ea1.buff.append(ea1.temp + "\r\n");
            System.out.println("Time to find Best Chromosome = " +
                    runFitnessTimes[numRuns - 1][2] +
                    " milliseconds; i.e., " +
                    (runFitnessTimes[numRuns - 1][2] / 1000) +
                    " seconds");
            ea1.temp = "Time to find Best Chromosome = " +
               runFitnessTimes[numRuns - 1][2] + " milliseconds; i.e., " +
               (runFitnessTimes[numRuns - 1][2] / 1000) + " seconds";
            ea1.buff.append(ea1.temp + "\r\n");
            System.out.println("Time to Complete EA run = " +
                    runFitnessTimes[numRuns - 1][3] +
                    " milliseconds; i.e., " +
                    (runFitnessTimes[numRuns - 1][3] / 1000) +
                    " seconds");
            ea1.temp = "Time to Complete EA run = " +
               runFitnessTimes[numRuns - 1][3] + " milliseconds; i.e., " +
               (runFitnessTimes[numRuns - 1][3] / 1000) + " seconds";
            ea1.buff.append(ea1.temp + "\r\n");
            System.out.println(" ");
            ea1.buff.append(ea1.temp + "\r\n\r\n");
            runFitnessTimes[numRuns -
               1][4] = ( ( (overall_BestofRuns - ea1.iLP_Opt) / ea1.iLP_Opt) *
                     100);
            System.out.println(" ");
            ea1.buff.append("\r\n");
            System.out.print("Run #" + numRuns + ". ");
            ea1.temp = "Run #" + numRuns + "). ";
            ea1.buff.append(ea1.temp);
            for (y = 0; y < 5; y++)
            {
             System.out.print(runFitnessTimes[numRuns - 1][y] + " | ");
             ea1.temp = runFitnessTimes[numRuns - 1][y] + " | ";
             ea1.buff.append(ea1.temp);
            }
            System.out.println(" ");
            ea1.buff.append("\r\n");
            System.out.println("(1: best fitness value, 2: gen best found, 3: time to best
(msec), 4: time to complete (msec), 5:% error)");
            ea1.buff.append("\r\n");
            System.out.println(" ");
            ea1.buff.append("\r\n");
            System.out.print("Best All Runs: ");
```

```
ea1.temp = "Best All Runs: ";
ea1.buff.append(ea1.temp + "\r\n");
System.out.println("Best Chromosome for Run Number #" + numRuns +
                ": ");
ea1.temp = "Best Chromosome for Run Number #" + numRuns + ": ";
ea1.buff.append(ea1.temp + "\r\n");
for (j = 0; j < ea1.IntNumContracts + 1; j++)
{
 System.out.print(ea1.bestChromosome[j] + " | ");
 ea1.temp = ea1.bestChromosome[j] + " | ";
 ea1.buff.append(ea1.temp);
}
System.out.println(" ");
ea1.buff.append("\r\n");
System.out.println("Best 20 chromosomes from EA run #" + numRuns +
                " for Problem # " + ea1.FileNum);
ea1.buff2.append("Best 20 chromosomes from EA run  \r\n");

double[][] tempA1 = new double[20][100];
StringBuffer sim_buff = new StringBuffer();
for(i=0;i<20;i++)
{
 tempA1[i]=ea1.training_results((double[]) ea1.Fill_Best.elementAt(i),i);
}
StringBuffer buff1 = new StringBuffer();
for(i=0;i<20;i++)
{
 for(k=0;k<100;k++)
 {
  buff1.append(tempA1[i][k]+",");
 }
 buff1.append("\r\n");
}
WriteFile sim_write = new WriteFile(buff1,ProblemNumber,"train_res");
sim_write.write_toFile();
double [] tempA = new double[ea1.IntNumContracts];
for (i = 0; i < 20; i++)
{
 tempA = (double[]) ea1.Fill_Best.elementAt(i);
 for (j = 0; j < tempA.length; j++)
 {
  ea1.temp2 = new Double(tempA[j]).toString() + " , ";
  ea1.buff2.append(ea1.temp2);
 }
 ea1.buff2.append("\r\n");
}
```

```
  ea1.buff2.append("\r\n");
  ea1.Fill_Best.removeAllElements();
  if (numRuns == 1)
  {
   ea1.write_to_file(true, "Best Chromosomes ");
  }
  else
  {
   ea1.write_to_file(false, "Best Chromosomes ");
  }
  System.out.println(" ");
  ea1.buff.append("\r\n");
}
System.out.println(" ");
ea1.buff.append("\r\n");
System.out.println("Total Time to Complete " + MaxRuns +
                   " EA runs = " +
                   TimetoCompleteforAllRuns +
                   " milliseconds; i.e., " +
                   (TimetoCompleteforAllRuns / 1000) + " seconds");
ea1.temp = "Total Time to Complete " + MaxRuns + " EA runs = " +
    TimetoCompleteforAllRuns + " milliseconds; i.e., " +
    (TimetoCompleteforAllRuns / 1000) + " seconds";
ea1.buff.append(ea1.temp + "\r\n");
System.out.println("Average Total Time to Complete " + MaxRuns +
                   " EA runs = " +
                   (TimetoCompleteforAllRuns / MaxRuns) +
                   " milliseconds; i.e., " +
                   (TimetoCompleteforAllRuns / MaxRuns / 1000) +
                   " seconds");
ea1.temp = "Average Total Time to Complete " + MaxRuns +
    " EA runs = " +
    (TimetoCompleteforAllRuns / MaxRuns) + " milliseconds; i.e., " +
    (TimetoCompleteforAllRuns / MaxRuns / 1000) + " seconds";
ea1.buff.append(ea1.temp + "\r\n");
System.out.println("Best Fitness Value found during " + MaxRuns +
                   " EA runs = " +
                   overall_BestofRuns);
ea1.temp = "Best Fitness Value found during " + MaxRuns +
    " EA runs = " +
    overall_BestofRuns;
ea1.buff.append(ea1.temp + "\r\n");

for (x = 1; x <= MaxRuns; x++)
{
  System.out.print("Run #" + x + ". ");
```

```
  ea1.temp = "Run #" + x + "). ";
  ea1.buff.append(ea1.temp + "\r\n");
  for (y = 0; y < 5; y++)
  {
    System.out.print(runFitnessTimes[x - 1][y] + " | ");
    ea1.temp = runFitnessTimes[x - 1][y] + " | ";
    ea1.buff.append(ea1.temp);
  }
  System.out.println(" ");
  ea1.buff.append("\r\n");
}
System.out.println(" ");
ea1.buff.append("\r\n");
System.out.print("Best All Runs: ");
ea1.temp = "Best All Runs: ";
ea1.buff.append(ea1.temp + "\r\n");
for (j = 0; j < ea1.IntNumContracts + 1; j++)
{
  System.out.print(ea1.Best_All_Runs[j] + " | ");
  ea1.temp = ea1.Best_All_Runs[j] + " | ";
  ea1.buff.append(ea1.temp);
}
System.out.println("\n ");
ea1.buff.append("\r\n\n");
System.out.println("Total Time to Complete All " + numRuns + "Runs: " +
                   TotalTimetoCompleteforAllRuns);
ea1.temp = "Total Time to Complete All " + numRuns + "Runs: " +
    TotalTimetoCompleteforAllRuns;
ea1.buff.append("\r\n");
System.out.println("D Array: ");
ea1.buff.append("D Array: ");
ea1.buff.append("\r\n");
for (j = 0; j < ea1.NumPeriods; j++)
{
  System.out.println(ea1.D[j]);
  ea1.temp = ea1.D[j] + "\n";
  ea1.buff.append(ea1.temp + "\r\n");
}
System.out.println("\n ");
ea1.buff.append("\r\n");


if (numRuns == 1)
{
  ea1.write_to_file(true, "");
}
```

```
          else
          {
           ea1.write_to_file(false, "");
          }
          Bingo = 0;

          if (ea1.gd >= .09)
          {
           if (z_count < 2)
           {
            ea1.gp = ea1.gp + increment_gp;
            ea1.gd = 0.03;
            z_count = z_count + 1;
           }
           else if (z_count >= 2)
           {
            z_count = 0;
            ea1.gp = 0.05;
            ea1.gd = 0.03;
           }
          }
          ea1.gd = ea1.gd + increment_gd;
          ea1.gd= new Double(Rounding.toString(ea1.gd,2)).doubleValue();
          ea1.gp = new Double(Rounding.toString(ea1.gp,2)).doubleValue();
        }
       }
      }
  }
}
```

Sim.java

```
package sim_ea;

import java.util.Random;
import java.math.*;

public class sim
{
 public Random random;
 public double demand[][];
 public double price[][];
 public double dem_prob []; //Probability of Demand Increase happening
 public double price_prob []; //Probability of Price Increase happening
 public int NumPeriod;  //Number of Periods
 public int NumSim; //Number of Simulations
 public double B[]; //Beta weights to calculate price
```

```
  public boolean dem_iid; //True demand increase is independent
  public boolean price_iid; //True price decrease in independent
  public boolean did_dem_increase[]; //For each instance did demand increase or decrease
in the previous period
  public boolean did_price_decrease[]; //For each instance did price decrease in the
previous period
  public int Problem_Number;

  public sim(int NumPeriod, double[] B, boolean dem_iid, boolean price_iid, int
ProblemNumber)
  {
   this.Problem_Number = ProblemNumber;
   this.price_iid = price_iid;
   this.dem_iid = dem_iid;
   this.NumPeriod = NumPeriod;
   this.B = B.clone();
   this.NumSim=500;
   this.random = new Random();
   this.demand = new double[this.NumPeriod][this.NumSim];
   this.price= new double [this.NumPeriod][this.NumSim];
   this.dem_prob = new double[this.NumSim];
   this.price_prob = new double[this.NumSim];
   this.did_dem_increase = new boolean[this.NumSim];
   this.did_price_decrease = new boolean[this.NumSim];
   for(int i = 0; i<this.NumSim;i++)
   {
    this.dem_prob[i]=0.5;
    this.price_prob[i]=0.5;
    this.did_dem_increase[i] = false;
    this.did_price_decrease[i] = false;
   }
  }

  public boolean bernoulli(double value)
  {
   boolean result = false;
   if(this.random.nextDouble()<value)
   {
    result = true;
   }
   return result;
  }

  public double exponential(double mean)
  {
   double value = 0;
```

```
  value = - mean * Math.log(1-this.random.nextDouble());
  return value;
 }

 public double uniform(double min, double max)
 {
  double value = this.random.nextDouble();
  value  = min + (max - min)*value;
  return value;
 }

 public double normal (double mean, double stdev)
 {
  double value = 0;
  value = mean + stdev*this.random.nextGaussian();
  return value;
 }

 public void write(int distribution)
 {
  StringBuffer dem_buff = new StringBuffer();
  StringBuffer price_buff = new StringBuffer();
  for(int i=0;i<this.NumPeriod;i++)
  {
   for (int j = 0; j < this.NumSim; j++)
   {
    dem_buff.append(Rounding.toString(this.demand[i][j],0)+",");
    price_buff.append(Rounding.toString(this.price[i][j],4)+",");
   }
   dem_buff.append("\r\n");
   price_buff.append("\r\n");
  }
  WriteFile dem_write = new WriteFile(dem_buff, this.Problem_Number,"Demand-
"+distribution+"-");
  dem_write.write_toFile();
  WriteFile price_write = new WriteFile(price_buff, this.Problem_Number,"Price-
"+distribution+"-");
  price_write.write_toFile();
 }

 public void populate(int distribution, double gp, double gd, double d_spread, double
p_spread)
 {
  for(int i=0;i<this.NumPeriod;i++)
  {
   for(int j=0;j<this.NumSim;j++)
```

```
{
 if(i==0)
 {
  this.demand[0][j]=540;
  this.price[0][j]=1;
  this.did_price_decrease[j] = false;
  this.did_dem_increase[j]=false;
  continue;
 }
 if(this.dem_iid==false)
 {
  if (this.did_dem_increase[j] == true)
  {
   this.dem_prob[j] = this.dem_prob[j];
  }
  else
  {
   this.dem_prob[j] = this.dem_prob[j] + 0.05;
  }
 }
 boolean dem_increase = this.bernoulli(this.dem_prob[j]);
 if (dem_increase == true)
 {
  if(distribution == 1)//1 for Uniform, 2 for Normal, 3 for Exponential
  {
   this.demand[i][j]=this.demand[i-1][j]*(1+this.uniform(gd,gd+d_spread));
  }
  else if (distribution == 2)
  {
   this.demand[i][j]=this.demand[i-1][j]*(1+this.normal(gd,d_spread));
  }
  else if (distribution ==3)
  {
   this.demand[i][j]=this.demand[i-1][j]*(1+this.exponential(gd));
  }
  this.did_dem_increase[i] = true;
 }
 else
 {
  this.demand[i][j]=this.demand[i-1][j];
  this.did_dem_increase[j] = false;
 }
 if(this.price_iid==false)
 {
  if (this.did_price_decrease[j] == true)
  {
```

```
          this.price_prob[j] = this.price_prob[j];
         }
         else
         {
          this.price_prob[j] = this.price_prob[j] + 0.05;
         }
        }
        boolean price_decrease = this.bernoulli(this.price_prob[j]);
        if (price_decrease == true)
        {
         if(distribution == 1)//1 for Uniform, 2 for Normal, 3 for Exponential
         {
          this.price[i][j]=this.price[i-1][j]*(1-this.uniform(gp,gp+p_spread));
         }
         else if (distribution == 2)
         {
          this.price[i][j]=this.price[i-1][j]*(1-this.normal(gp,p_spread));
         }
         else if (distribution ==3)
         {
          this.price[i][j]=this.price[i-1][j]*(1-this.exponential(gp));
         }
         this.did_price_decrease[i] = true;
        }
        else
        {
         this.price[i][j]=this.price[i-1][j];
         this.did_price_decrease[j] = false;
        }
       }

    }
    this.write(distribution);
  }
}
```

Testing

Start.java

```
package sim_contract;
import java.util.StringTokenizer;
import java.util.Vector;
public class start
{
```

```
  public ReadFile read;
  public WriteFile write;
  public int NumPeriods;
  public int NumContracts;
  public int ProblemNumber;
  public double[] B;
  public double [] D;
  public double [] rp_Array;
  public double [] rd_Array;
  public double gp;
  public double gd;
  public sim sim1;
  public Vector solution;
  public int [] Contract_Period;
  public int [] Contract_Length;
  /************SIM VARIABLES*****************************************/
  public double sim_demand[][];
  public double sim_price[][];
  public double[][] sim_chromosome_value;
  public int sim_instances;

/*********************************************************************/

  public start ()
  {
   this.B = new double[3];
   this.solution = new Vector();
  }

  /*Updating demand array with average demand from simulation*****************/
  public void sim_update_demand(double[][] demand)
  {
   double value = 0;
   for(int i=0;i<this.NumPeriods;i++)
   {
    for(int j=0;j<demand[i].length;j++)
    {
     value = value + demand[i][j];
    }
    this.D[i] = value/demand[i].length;
    value=0;
   }
  }
  /*Updating price array with average price from simulation******************/
  public void sim_update_price(double[][] price)
  {
```

```
  double value = 0;
  for(int i=0;i<this.NumPeriods;i++)
  {
   for(int j=0;j<price[i].length;j++)
   {
    value = value + price[i][j];
   }
   this.rp_Array[i] = value/price[i].length;
   value=0;
  }
}

public void initialize_globalvariables()
{
  this.D = new double[this.NumPeriods];
  this.rp_Array = new double[this.NumPeriods];
  this.rd_Array = new double[this.NumPeriods];
  this.NumContracts=this.NumPeriods*(this.NumPeriods+1)/2;
  this.Contract_Length = new int[this.NumContracts];
  this.Contract_Period= new int[this.NumContracts];
  int index = 0;
  for (int i = 0; i <= NumPeriods; i++)
  {
   for (int j = 1; j <= NumPeriods - i; j++)
   {
    Contract_Length[index] = j;
    index = index + 1;
   }
  }
  int count = 0;
  for (int i = 1; i < this.NumPeriods + 1; i++)
  {
   for (int j = 1; j < this.NumPeriods + 2 - i; j++)
   {
    this.Contract_Period[count] = i;
    count = count + 1;
   }
  }
}

public void parse_input()
{
  StringTokenizer token = new StringTokenizer(this.read.buf.toString(),",");
  StringBuffer buff = new StringBuffer();
  double[] chromosome = new double[this.NumContracts + 1];
  for(int i=0;i<20;i++)
```

```
  {
   for(int j=0;j<=this.NumContracts;j++)
   {
    chromosome[j]=new Double(token.nextToken()).doubleValue();
   }
   buff.append(chromosome[0]+"\r\n");
   this.solution.addElement((double[])chromosome.clone());
  }
  WriteFile write = new WriteFile(buff,1,"Det-fitness");
  try
  {
     write.append_toFile();
  }
  catch(Exception ex)
  {
     System.out.println(ex);
  }
}

public Vector valid_contracts(int timeperiod)
{
   Vector valid = new Vector();
   int x = this.NumPeriods;
   int y = timeperiod;
   for (int i =1;i<=timeperiod;i++)
   {
      for(int j=x;j>=y;j--)
      {
         valid.addElement(new Integer(j));
      }
      x= x+this.NumPeriods-i;
      y=y+this.NumPeriods -i;
   }
   valid.trimToSize();
   return valid;
  }

public double find_deficit(double[] chromosome, int Period, int instance)
{
  double deficit = 0;
  double contract_amount = 0;
  Vector temp = this.valid_contracts(Period+1);
  for(int i =0;i<temp.size();i++)
  {
   int index = ((Integer)temp.elementAt(i)).intValue();
   contract_amount = contract_amount+chromosome[index];
```

```
    }
    if(instance >=0)
    {
     deficit = this.sim_demand[Period][instance] - contract_amount;
    }
    else
    {
     deficit = this.D[Period] - contract_amount;
    }
    if(deficit<0)
     deficit = 0;
    return deficit;
  }
  public double [] training_results(double[] chromosome, int chromosome_index)
  {
    int IntNumContracts = NumContracts;
    double [] sim_evaluation = new double[this.sim_instances];
    double avgTotal = 0;
    for(int i=0;i<this.NumPeriods;i++)
    {
     for(int j=0;j<this.sim_instances;j++)
     {
      double deficit = this.find_deficit(chromosome,i,j);
      if (deficit !=0)
      {
       sim_evaluation[j]=sim_evaluation[j]+deficit*(this.sim_price[i][j]*
          B[0]*1+B[2])+this.sim_price[i][j]*
          B[1]*1;
      }
     }

    }
    for (int i=0;i<IntNumContracts;i++)
    {
     int contract_period = this.Contract_Period[i];
     int contract_length = this.Contract_Length[i];
     double contract_amount = chromosome[i+1];
     for(int j =0;j<this.sim_instances;j++)
     {
      if(contract_amount != 0)
      {

sim_evaluation[j]=sim_evaluation[j]+contract_amount*(this.sim_price[contract_period-
1][j]*
          B[0]*contract_length+B[2])+this.sim_price[contract_period-1][j]*
          B[1]*contract_length;
```

```
      }
    }
  }
  return sim_evaluation;
  }


public static void main ( String[] args )
{
  start start1 = new start () ;
  for (int x_n = 1; x_n <= 3; x_n++)
  {
    if (x_n==1)
    {
      start1.NumPeriods=6;
    }
    else if (x_n==2)
    {
      start1.NumPeriods=12;
    }
    else if(x_n==3)
    {
      start1.NumPeriods=18;
    }
    else
    {
      System.exit(0);
    }
    start1.initialize_globalvariables();
    for (int x_Size = 1; x_Size <= 3; x_Size++)
    {
      start1.gp = .05;
      start1.gd = .05;
      //Commented for testing deterministic sol


      if (x_Size == 1)
      {
        //Small economies of scale
        /* */
        start1.B[0] = 13.116;
        start1.B[1] = 962.927;
        start1.B[2] = 258.526;
        //econScale = "econScale-Small";
        /* */
      }
```

```
        if (x_Size == 2)
        {
         //Medium economies of scale
         /* */
         start1.B[0] = 6.757;
         start1.B[1] = 928.503;
         start1.B[2] = 364.989;
         //econScale = "econScale-Medium";
         /* */
        }
        if (x_Size == 3)
        {
         //Large economies of scale
         /* */
         start1.B[0] = 0.3977;
         start1.B[1] = 894.115;
         start1.B[2] = 471.416;
         //econScale = "econScale-Large";
         /* */
        }
        int counter =0;
        for (int numProb = 1; numProb <= 9; numProb++)
        {
         start1.solution = new Vector();
         start1.ProblemNumber++;

/*****************************************************************/
/**********************SIMULATION CODE**************************/

/*****************************************************************/
          start1.sim1 = new sim(start1.NumPeriods,start1.B, false,
false,start1.ProblemNumber);
          int distribution = 1;//1 for Uniform, 2 for Normal, 3 for Exponential
          double d_spread = 0.1;// This gives the width for Uniform and std dev for Normal
distribution (< 1)
          double p_spread = 0.1;

          if(start1.gd == 0.05)
          {
           d_spread = 0.1;
          }
          else if (start1.gd == 0.07)
          {
           d_spread = 0.06;
          }
          else if (start1.gd == 0.09)
```

```
        {
         d_spread = 0.02;
        }
        if(start1.gp == 0.05)
        {
         p_spread = 0.1;
        }
        else if (start1.gp == 0.07)
        {
         p_spread = 0.06;
        }
        else if (start1.gp == 0.09)
        {
         p_spread = 0.02;
        }
        start1.sim1.populate(distribution,start1.gp,start1.gd,d_spread, p_spread);
        start1.sim_price = start1.sim1.price.clone();
        start1.sim_demand = start1.sim1.demand.clone();
        start1.sim_update_demand(start1.sim_demand);
        start1.sim_update_price(start1.sim_price);
        start1.sim_instances = start1.sim1.NumSim;

/*****************************************************************/
        start1.read = new ReadFile("Best Chromosomes "+start1.ProblemNumber+".txt");
        try
        {
         start1.read.read_input();
        }
        catch(Exception ex)
        {
         System.out.println(ex);
        }
        start1.parse_input();
        System.out.println("Best Chromosomes "+start1.ProblemNumber+".txt "+
              "gd="+start1.gd+" gp="+start1.gp+" d_spread="+d_spread+"
p_spread="+p_spread);
        double[][] tempA = new double[20][100];
        StringBuffer sim_buff = new StringBuffer();
        for(int i=0;i<20;i++)
        {
         double[] temp =(double[]) start1.solution.elementAt(i);
         tempA[i]=start1.training_results(temp,i);
        }
        StringBuffer buff1 = new StringBuffer();
        for(int i=0;i<20;i++)
        {
```

```
            for(int j=0;j<100;j++)
            {
              buff1.append(tempA[i][j]+",");
            }
            buff1.append("\r\n");
          }
          WriteFile sim_write = new WriteFile(buff1,start1.ProblemNumber,"test_res");
          sim_write.write_toFile();
          start1.gd=new Double(Rounding.toString(start1.gd+0.02,2)).doubleValue();
          counter++;
          if(counter > 2)
          {
            start1.gd=0.05;
            start1.gp=new Double(Rounding.toString(start1.gp+0.02,2)).doubleValue();
            counter =0;
          }
        }
      }
    }
  }
}
```

Sim.java

```
package sim_contract;
import java.util.Random;
import java.math.*;

public class sim
{
  public Random random;
  public double demand[][];
  public double price[][];
  public double dem_prob []; //Probability of Demand Increase happening
  public double price_prob []; //Probability of Price Increase happening
  public int NumPeriod;  //Number of Periods
  public int NumSim; //Number of Simulations
  public double B[]; //Beta weights to calculate price
  public boolean dem_iid; //True demand increase is independent
  public boolean price_iid; //True price decrease in independent
  public boolean did_dem_increase[]; //For each instance did demand increase or decrease
in the previous period
  public boolean did_price_decrease[]; //For each instance did price decrease in the
previous period
  public int Problem_Number;
```

```
 public sim(int NumPeriod, double[] B, boolean dem_iid, boolean price_iid, int
ProblemNumber)
 {
  this.Problem_Number = ProblemNumber;
  this.price_iid = price_iid;
  this.dem_iid = dem_iid;
  this.NumPeriod = NumPeriod;
  this.B = B.clone();
  this.NumSim=100;
  this.random = new Random();
  this.demand = new double[this.NumPeriod][this.NumSim];
  this.price= new double [this.NumPeriod][this.NumSim];
  this.dem_prob = new double[this.NumSim];
  this.price_prob = new double[this.NumSim];
  this.did_dem_increase = new boolean[this.NumSim];
  this.did_price_decrease = new boolean[this.NumSim];
  for(int i = 0; i<this.NumSim;i++)
  {
   this.dem_prob[i]=1;
   this.price_prob[i]=1;
   this.did_dem_increase[i] = false;
   this.did_price_decrease[i] = false;
  }
 }

 public boolean bernoulli(double value)
 {
  boolean result = false;
  if(this.random.nextDouble()<value)
  {
   result = true;
  }
  return result;
 }

 public double exponential(double mean)
 {
  double value = 0;
  value = - mean * Math.log(1-this.random.nextDouble());
  return value;
 }

 public double uniform(double min, double max)
 {
  double value = this.random.nextDouble();
  value  = min + (max - min)*value;
```

```
  return value;
 }

 public double normal (double mean, double stdev)
 {
  double value = 0;
  value = mean + stdev*this.random.nextGaussian();
  return value;
 }

 public void write(int distribution)
 {
  StringBuffer dem_buff = new StringBuffer();
  StringBuffer price_buff = new StringBuffer();
  for(int i=0;i<this.NumPeriod;i++)
  {
   for (int j = 0; j < this.NumSim; j++)
   {
    dem_buff.append(Rounding.toString(this.demand[i][j],0)+",");
    price_buff.append(Rounding.toString(this.price[i][j],4)+",");
   }
   dem_buff.append("\r\n");
   price_buff.append("\r\n");
  }
  WriteFile dem_write = new WriteFile(dem_buff, this.Problem_Number,"Demand-
"+distribution+"-");
  dem_write.write_toFile();
  WriteFile price_write = new WriteFile(price_buff, this.Problem_Number,"Price-
"+distribution+"-");
  price_write.write_toFile();
 }

 public void populate(int distribution, double gp, double gd, double d_spread, double
p_spread)
 {
  for(int i=0;i<this.NumPeriod;i++)
  {
   for(int j=0;j<this.NumSim;j++)
   {
    if(i==0)
    {
     this.demand[0][j]=540;
     this.price[0][j]=1;
     this.did_price_decrease[j] = false;
     this.did_dem_increase[j]=false;
     continue;
```

```
}
if(this.dem_iid==false)
{
 if (this.did_dem_increase[j] == true)
 {
  this.dem_prob[j] = this.dem_prob[j];
 }
 else
 {
  this.dem_prob[j] = this.dem_prob[j] + 0.05;
 }
}
boolean dem_increase = this.bernoulli(this.dem_prob[j]);
if (dem_increase == true)
{
 if(distribution == 1)//1 for Uniform, 2 for Normal, 3 for Exponential
 {
  this.demand[i][j]=this.demand[i-1][j]*(1+this.uniform(gd,gd+d_spread));
 }
 else if (distribution == 2)
 {
  this.demand[i][j]=this.demand[i-1][j]*(1+this.normal(gd,d_spread));
 }
 else if (distribution ==3)
 {
  this.demand[i][j]=this.demand[i-1][j]*(1+this.exponential(gd));
 }
 this.did_dem_increase[i] = true;
}
else
{
 this.demand[i][j]=this.demand[i-1][j];
 this.did_dem_increase[j] = false;
}
if(this.price_iid==false)
{
 if (this.did_price_decrease[j] == true)
 {
  this.price_prob[j] = this.price_prob[j];
 }
 else
 {
  this.price_prob[j] = this.price_prob[j] + 0.05;
 }
}
boolean price_decrease = this.bernoulli(this.price_prob[j]);
```

```
        if (price_decrease == true)
        {
          if(distribution == 1)//1 for Uniform, 2 for Normal, 3 for Exponential
          {
            this.price[i][j]=this.price[i-1][j]*(1-this.uniform(gp,gp+p_spread));
          }
          else if (distribution == 2)
          {
            this.price[i][j]=this.price[i-1][j]*(1-this.normal(gp,p_spread));
          }
          else if (distribution ==3)
          {
            this.price[i][j]=this.price[i-1][j]*(1-this.exponential(gp));
          }
          this.did_price_decrease[i] = true;
        }
        else
        {
          this.price[i][j]=this.price[i-1][j];
          this.did_price_decrease[j] = false;
        }
       }
     }
    this.write(distribution);
  }
}
```